A UNIX operációs rendszer

Brian W. Kernighan
Rob Pike

Műszaki Könyvkiadó

Az eredeti mű:
B. W. Kernighan – R. Pike: The UNIX Programming Environment
Original English language edition published by Copyright © 1984 by Prentice-Hall Inc.
All Rights Reserved

Lektorálta:
Sztrókay Kálmán
okl. matematika- és fizikatanár

A verseket **Tandori Dezső** fordította.

©Hungarian translation Turi Gabriella, Kovács Tibor, 1987

ETO: 519.68

ISBN: 963 10 7452 8

Ezzel a könyvvel az OKKFT G-1-es programját támogatjuk.

Tartalomjegyzék

Előszó 9

1. UNIX-ról a kezdőknek 13
   - 1.1. A kezdet: néhány szó a terminálokról és a gépelésről 14
   - 1.2. Mindennapos eszközök: fájlok, általános parancsok 23
   - 1.3. Még valami a fájlokról: a jegyzékek 34
   - 1.4. A burok 38
   - 1.5. Ami a UNIX rendszerből még hátravan 52

2. A fájlrendszer 54
   - 2.1. A fájlok alapjai 54
   - 2.2. Mi van a fájlban? 58
   - 2.3. Jegyzékek és fájlnevek 61
   - 2.4. Engedélyek 64
   - 2.5. Az i-bögök 70
   - 2.6. A jegyzékhierarchia 75
   - 2.7. Eszközök 78

3. A burok használata 83
   - 3.1. A parancssor szerkezete 83
   - 3.2. Metakarakterek 86
   - 3.3. Új parancsok létrehozása 93
   - 3.4. Parancsargumentumok és paraméterek 95
   - 3.5. Programkimenet mint argumentum 99
   - 3.6. Burokváltozók 101
   - 3.7. Még többet az átirányításról 105
   - 3.8. Burok-programciklusok 108
   - 3.9. Kössünk össze mindent: **bundle** 110
   - 3.10. Miért programozható burok? 112

4. Szűrők 115
   - 4.1. A **grep** család 116
   - 4.2. További szűrők 120

4.3. A **sed** áradatszerkesztő 122
4.4. Az **awk** mintakereső és -feldolgozó nyelv 129
4.5. Jó fájlok és jó szűrők 146

5. Burokprogramozás 148
5.1. A **cal** parancs átalakítása 148
5.2. Melyik parancs az igazi: **which**? 153
5.3. Változások figyelése: **while** és **until** 159
5.4. Csapdák: megszakítások elfogása 164
5.5. Fájlcsere: **overwrite** 167
5.6. Folyamatok név szerinti leállítása: **zap** 171
5.7. Szóközök és argumentumok kezelése: **pick** 173
5.8. Közhasznú üzenetek: **news** 176
5.9. Fájlváltozások követése: **get** és **put** 179
5.10. Visszatekintés 184

6. Programozás a szabványos be- és kimenettel 186
6.1. Szabványos bemenet és kimenet: **vis** 187
6.2. Programargumentumok: a **vis** második változata 190
6.3. Fájlelérés: a **vis** harmadik változata 192
6.4. Képernyőnyomtató: **p** 196
6.5. Egy példa: **pick** 202
6.6. Hibák és nyomkeresés 203
6.7. Egy példa: **zap** 205
6.8. Egy interaktív fájl-összehasonlító program: **idiff** 208
6.9. A környezet elérése 213

7. Rendszerhívások 216
7.1. Alacsony szintű be- és kimenet 216
7.2. Fájlrendszer; jegyzékek 223
7.3. Fájlrendszer; i-bögök 228
7.4. Folyamatok 233
7.5. Jelek és megszakítások 239

8. Programfejlesztés 245
8.1. Első lépcső: a négyműveletes számológép 246
8.2. Második lépcső: változók és hibakezelés 254
8.3. Harmadik lépcső: tetszőleges változónevek; beépített függvények 258
8.4. Negyedik lépcső: gépre vitel 269
8.5. Ötödik lépcső: programvezérlés; relációs operátorok 277
8.6. Hatodik lépcső: függvények és eljárások; be- és kimenet . . . 284
8.7. A hatékonyság vizsgálata 294
8.8. Visszatekintés 296

9. Dokumentumok készítése 299
9.1. Az **ms** makrocsomag 300
9.2. A **troff** szintje 308

9.3. A **tbl** és az **eqn** preprocesszorok 312
9.4. A kézikönyvlap 320
9.5. Más dokumentumkészítő eszközök 325

10. Epilógus 327

1. függelék: Az editor összefoglalása 330
2. függelék: A **hoc** kézikönyv 342
3. függelék: A **hoc** lista 347

Tárgymutató 359

Előszó

„A UNIX rendszerek száma tízre nőtt és még több várható".
(A *UNIX Programozói kézikönyv* második kiadásából, 1972. június.)

A UNIX* operációs rendszer először egy kimustrált DEC PDP–7-en működött a Bell Laboratories-ben, 1969-ben. Ken Thompson – Rudd Canaday, Doug McIlroy, Joe Ossana és Dennis Ritchie elképzeléseivel és segítségével – írt egy kisméretű általános célú időosztásos rendszert, amely elég kényelmes volt ahhoz, hogy megnyerje a lelkes felhasználókat, és végül lehetőséget teremtett egy nagyobb gép (PDP–11/20) vásárlásához. Az egyik első felhasználó Ritchie volt, aki 1970-ben segített áttenni a rendszert PDP–11-re. Ritchie tervezett és írt a C programnyelvhez is egy fordítóprogramot. 1973-ban Ritchie és Thompson C nyelven újraírták a UNIX *magot* (kernel), ezzel megtörték azt a hagyományt, hogy a rendszerszoftvert assembly nyelven írják. Ez a rendszer már lényegében az volt, amit ma is használunk.

A UNIX 1974-ben az egyetemek számára oktatási célokra és néhány év múlva kereskedelmi forgalomban is hozzáférhető lett. Ebben az időszakban terjedt el a Bell Laboratories-nél, használták laboratóriumokban, szoftverfejlesztésre, szövegfeldolgozó rendszerekben, telefontársaságok működését szolgáló rendszerekben. Azóta világszerte sok tízezer gépen fut UNIX rendszer.

A UNIX sikerének számos oka van. Először is, mivel C-ben íródott, futtatható a mikroszámítógéptől kezdve bármely kiépítésű számítógépen, ami komoly kereskedelmi előny. Másodszor, a hozzáférhető és magas szintű nyelven íródott forráskód egyszerűvé teszi a rendszernek még a különleges követelményekhez való illesztését is. Végül, ami a leglényegesebb, a UNIX jó operációs rendszer, különösen a programozók számára. A UNIX programozási környezet szokatlanul gazdag és termelékeny.

Annak ellenére, hogy a UNIX rendszer számtalan újszerű programot és módszert tartalmaz, nem egy adott program vagy ötlet teszi jól működővé, hanem a programozás újszerű megközelítése, a számítógép használatának filozófiája. Bár ez a filozófia nem írható le egy mondatban, lényege az, hogy a rendszer ereje a programok egymás közti kapcsolatából származik és nem magukból a programokból. Sok UNIX program önmagában rendkívül egyszerű feladatokat valósít meg, de más programokkal összekapcsolva általános és hasznos eszközzé válik.

Könyvünkben ezzel a filozófiával akarjuk megismertetni az olvasót. Mivel a filozófia a programok kapcsolatára épül, mi itt leginkább az önálló eszközöket vizsgáljuk, közben végigmegyünk a programok összekapcsolásának kérdésein és

* A UNIX név a Bell Laboratories védjegye.

azon, hogy hogyan használjunk programokat újabb programok építésére. Nemcsak az önálló programokat, hanem környezetükhöz illeszkedésüket is értenünk kell ahhoz, hogy jól használjuk a UNIX rendszert és elemeit.

Ahogy terjedt a UNIX rendszer, úgy csökkent azoknak a száma, akik annak alkalmazásában képzettek. Gyakran találkozunk olyan tapasztalt felhasználókkal (magunkat is beleértve), akik csak ügyetlen megoldást találnak egy-egy problémára, programokat írnak olyan feladatokra, amelyek meglévő eszközökkel könnyen kezelhetők. Az igazán jó megoldásokat természetesen nem könnyű tapasztalat és hozzáértés nélkül megtalálni. Reméljük a könyv segít abban, hogy kezdő és haladó Olvasóink mélyebben megértsék és hatásosan; élvezettel alkalmazzák a rendszert.

Könyvünk abban a reményben szól a programozókhoz, hogy az ő munkájukat eredményesebbé téve a munkacsoportok munkája is eredményesebb lesz. Annak ellenére, hogy feltételezünk programozási ismereteket, az első négy-öt fejezet megértéséhez nincs szükség programozói tapasztalatra, így az az egyéb felhasználók számára is hasznos lehet.

Ahol lehetett, állításainkat igyekeztünk valóságos példákkal illusztrálni. Néhány, a könyvhöz példaként született program azóta mindennapi programkészletünk része. A példákat közvetlenül a könyv számítógép által olvasható szövegéből teszteltük. A könyv felépítése a következő:

Az első fejezet bevezetés a rendszer alapvető használatába. Magában foglalja a bejelentkezést, a levelezést, a fájlrendszert, az általánosan használt parancsokat, a parancsértelmezővel kapcsolatos alapfogalmakat. Tapasztalt felhasználók ezt a fejezetet kihagyhatják.

A második fejezet a UNIX fájlrendszer leírása. A fájlrendszer alapvető a rendszer működése és használata szempontjából. Ezt kell érteni ahhoz, hogy jól használhassuk a rendszert. Itt ismertetjük a fájlokat, a jegyzékeket,* az engedélyeket, a fájlmódokat és az i-bögöket. Végül foglalkozunk a fájlrendszer-rangsorral és az eszközfájlokkal.

A harmadik fejezetben a programfuttatás és a programírás alapvető eszközével, a parancsértelmezővel vagy *burokkal* (*shell*) foglalkozunk. Leírjuk, hogyan használhatjuk saját céljainkra, új parancsok, parancsváltozók, burokváltozók, alapvető vezérlőfolyamatok létrehozására, a be- és kimenet (I/O) átirányítására.

A negyedik fejezet a *szűrőkről* (*filters*) szól, ezek olyan programok, amelyek egyszerű transzformációkat hajtanak végre a rajtuk átfolyó adatokon. Az első szakasz a **grep** mintakereső paranccsal és rokonaival foglalkozik, a következő bemutat néhány gyakran használt szűrőt, ilyen pl. a **sort.** A fejezet többi részét két általános adatátalakító programnak szenteltük, ezek a **sed** és az **awk.** A **sed** áradatszerkesztő segítségével javításokat, változtatásokat hajthatunk végre a rajta átfolyó adatáradaton. Az **awk** egyszerű információ-visszanyerő és listázási feladatokra alkalmazható programnyelv. E programok alkalmazásával, néha a burokkal együttműködve gyakran feleslegessé válik a hagyományos programozás.

Az ötödik fejezetben megtudjuk, hogyan használhatjuk a burkot, ha mások által használandó programokat írunk. Foglalkozunk bonyolultabb vezérlőfolyamatokkal, változókkal, szólunk a *csapdákról* (*trap*) és a megszakítások kezeléséről. Példáink bemutatják a **sed,** az **awk** és a burok nevezetes alkalmazásait.

* A UNIX directory fogalma más rendszerek katalógusfogalmától lényegesen különbözik, ennek hangsúlyozására választottuk a jegyzék elnevezést. *(A lektor.)*

A hatodik fejezetben a szabványos be- és kimenet könyvtárat felhasználó új programok írásával foglalkozunk, hiszen a burokkal és a meglevő programokkal nem lehet mindent megvalósítani. A szabványos könyvtár programjait C nyelven írták, s feltételezzük, hogy ezt a nyelvet az olvasó ismeri vagy a könyv olvasásával párhuzamosan tanulja. Igyekeztünk értelmes stratégiákat bemutatni az új programok tervezésére és szervezésére, az új program fokozatos felépítésére, a meglévő eszközök felhasználásának módjára.

A hetedik fejezet a rendszerhívásokkal, azaz az összes többi szoftverréteg alapzatával foglalkozik. A kérdéskörhöz tartozik a be- és kimenet, a fájllétrehozás, a hibakezelés, a katalógusok, az i-bögök, a folyamatok és a jelek.

A nyolcadik fejezet a programfejlesztő eszközökről, az **yacc** elemzőgenerátorról, a nagy programok fordítását ellenőrző **make**-ről, a szókészletelemzőket generáló **lex**-ről szól. Ezeket egy nagy programnak, egy C nyelvhez hasonló programozható kalkulátor fejlesztésén mutatjuk be.

A kilencedik fejezet a dokumentációt készítő eszközöket a nyolcadik fejezetben szereplő kalkulátorhoz készített felhasználói szintű leírás és UNIX kézikönyvlap segítségével mutatja be. Ez a fejezet a többi fejezettől függetlenül is olvasható.

Az első függelék összefoglalja az **ed** szabványos szövegszerkesztő programot. Bár a felhasználók egy része a mindennapi munkában más szerkesztőprogramot részesít előnyben, az **ed** mindig elérhető, hatékony és termelékeny. Már csak azért is érdemes megtanulni, mert szabályos kifejezései más programok, pl. a **grep** és a **sed** magjai is.

A második függelék a nyolcadik fejezet kalkulátornyelvének kézikönyve.

A harmadik függelék a kalkulátorprogram utolsó változatának listája. A kényelmes olvashatóság céljából tartalmazza a kódokat is.

Néhány gyakorlati tanács:

1. A UNIX rendszer nagyon népszerűvé vált, és sok változatát használják. Például a 7. kiadás a UNIX rendszernek a Bell Laboratories Computing Science Research Centerjében használt eredeti forráskódjából származott. A System III. és a System, V. a Bell Laboratories által hivatalosan támogatott változatok. A Berkeleyi University of California egy, a 7. kiadásból levezetett változatot forgalmaz, ezt UCB 4.xBSD néven ismerjük. Ezenkívül különösen kisszámítógépeken számos, a 7. kiadásból származó változat létezik. Ezt a sokféleséget úgy próbáltuk átfogni, hogy azokat a vonatkozásokat vizsgáljuk, amelyek valószínűleg minden rendszerben azonosak. Bár amit ismertünk, az független minden egyedi rendszertől; egyes részleteket úgy mutatunk be, ahogyan azok a 7. kiadásban szerepelnek, miután ez az alapja a széles körben elterjedt UNIX rendszereknek.

A példákat lefuttattuk a Bell Laboratories System V. és a Berkeley 4.1BSD rendszerén is, ami csak néhány példában igényelt kisebb változtatásokat. Bármelyik változatot használhatjuk, a különbségek jelentéktelenek.

2. Ez a könyv nem referenciakönyv, bár nagyon sok anyag van benne. Fontosabbnak éreztük a megközelítés, az alkalmazási stílus tanítását, mint a részleteket. Az információk pontos forrása a *UNIX Programozói kézikönyv.** Szükség lesz rá, hogy olyan részleteket megvilágítson, amelyekkel mi nem foglalkozunk, és hogy meghatározza az Olvasó rendszere és a mienk közötti különbségeket.

* A UNIX kézikönyv (UNIX Programmer's Manual) aktuális változata általában megtalálható a UNIX-szal működő rendszerek mellett.

3. Hisszük, hogy a tanulás legjobb módja, ha csináljuk is, amit tanulunk. Ezt a könyvet egy terminálnál kell olvasni, így állításaink ellenőrizhetők, és felderíthetők a korlátok és a változatok. Olvassunk el egy részt a könyvből, próbáljuk ki, azután olvassunk tovább!

Hisszük, hogy a UNIX rendszer csodálatos számítógépes környezet, bár bizonyosan nem tökéletes. Reméljük, hogy a könyvet elolvasva Olvasóink is hasonló következtetésre jutnak.

Hálásak vagyunk mindazoknak, akik építő hozzászólásaikkal, kritikával segítették tökéletesíteni a szöveget, különösen John Bentleynek, John Lindermannak, Doug McIlroynak és Peter Weinbergernek, akik különös gonddal olvasták a kézirat változatait. Lekötelezettjei vagyunk Al Ahónak, Ed Bradfordnak, Bob Flandrenának, Dave Hansonnak, Ron Hardinnak, Larry Roslernak, Chris Van Wyknek és Jim Weythmannak a kézirat első változatához fűzött megjegyzéseikért. Köszönjük Mike Bianchi, Elizabeth Bimmler, Joe Carfagno, Don Carter, Tom De Marco, Tom Duff, David Gay, Steve Mahaney, Ron Pinter, Dennis Ritchie, Ed Sitar, Ken Thompson, Mike Tilson, Paul Tukey, Larry Werr értékes javaslatait.

Brian Kernighan
Rob Pike

1. UNIX-ról a kezdőknek

Mi az a UNIX? A legszűkebb értelemben egy időosztásos operációs rendszer *magja* (kernel): egy program, amely vezérli a számítógép erőforrásait és elosztja azokat a felhasználók között. Lehetővé teszi a felhasználók számára a programok futtatását, vezérli a géphez kapcsolt perifériákat (mágneslemezek, terminálok, nyomtatók stb.), fájlrendszert biztosít az adatok, programok, dokumentumok tárolására.

Tágabb értelemben nemcsak a magot nevezzük UNIX-nak, hanem beleértjük az olyan fontos programokat, mint a fordítók, szövegszerkesztők, parancsnyelvek, fájlokat másoló és kinyomtató programok.

Még tágabb értelemben a UNIX tartalmazza az általunk vagy más felhasználók által fejlesztett, a rendszerünkön futó programokat, pl. dokumentumkészítő eszközöket, statisztikai elemzőrutinokat, grafikus csomagokat.

Az, hogy éppen melyik értelemben használjuk a UNIX nevet, attól függ, hogy a rendszer melyik szintjét vizsgáljuk. Könyvünk további részében a szövegkörnyezetből fog kiderülni, hogy a UNIX elnevezést éppen melyik értelemben használjuk.

A UNIX rendszer néha bonyolultabbnak tűnik, mint amilyen valójában – a kezdő nehezen ismeri fel, hogy hogyan használhatja ki legjobban a rendelkezésre álló lehetőségeket. Szerencsére nem nehéz a kezdet, néhány program ismerete elég az induláshoz. E fejezet célja, hogy a kezdő minél gyorsabban használhassa a rendszert. Itt csak áttekintést adunk, nem kézikönyvet, az anyag nagy részét a későbbi fejezetekben részletesebben tárgyaljuk. Főleg a következő területekről fogunk beszélni:

- alapok – be- és kijelentkezés, egyszerű parancsok, gépelési hibák javítása, levelezés, terminálok közötti kapcsolatteremtés;
- mindennapi használat – fájlok, a fájlrendszer, a fájlok kinyomtatása, katalógusok, az általánosan használt parancsok;
- a parancsértelmező vagy *burok* (shell) – fájlnévrövidítések, a kimenet és bemenet átirányítása, csőhálózatok, törlő- és sorirtó karakterek beállítása, saját parancskereső út beállítása.

Annak, aki már használta a UNIX rendszert, a fejezet legnagyobb része ismerős lesz, s ha tetszik, átugorhat a második fejezetre.

Már e fejezet olvasásakor is legyen kéznél a *UNIX Programozói kézikönyv:* ahelyett, hogy itt megismételnénk az abban foglaltakat, gyakran fogjuk kérni, hogy egy adott részt nézzenek meg abban.

Ez a könyv nem helyettesíti a kézikönyvet, hanem azt mutatja meg, hogy hogyan

lehet a legjobban használni az abban leírt parancsokat. Továbbá lehet, hogy amit itt mondunk, az nem pontosan igaz arra a rendszerre, amellyel Olvasónk dolgozik. A kézikönyv elején lévő tárgymutató nélkülözhetetlen ahhoz, hogy problémánk megoldásához megtaláljuk a megfelelő programokat; tanuljuk meg használni.

Végül egy tanács, ne féljünk kísérletezni! A kezdő nemigen tud olyan hibát csinálni, amivel kárt okozna magának vagy a többi felhasználónak. Próbálgatva tanuljuk meg a dolgokat! Ez a fejezet hosszú, s legjobb, ha néhány oldalt elolvasunk, kipróbáljuk a dolgokat, azután haladunk tovább.

1.1. A kezdet: néhány szó a terminálokról és a gépelésről

Feltételezzük, hogy az Olvasónak van valamilyen ismerete a számítógép-terminálokról és azok használatáról. Ha a következő kijelentések közül valamelyik nem világos, kérjen segítséget a gépnél levő szakembertől.

A UNIX rendszer *kétirányú* (full duplex): a billentyűzeten leütött karakterek továbbítódnak a rendszerhez, amely visszaküldi őket a terminálra és megjelennek a képernyőn. Ez a *visszhang* (echo) folyamat a leütött karaktereket közvetlenül a képernyőre másolja, így látjuk, amit begépelünk. Van, amikor a visszhang kikapcsolódik, pl. a titkos jelszó beírásakor a karakterek nem jelennek meg a képernyőn.

A billentyűzeten levő karakterek legtöbbje közönséges kiírható karakter, nincs semmilyen különleges jelentése. Néhány billentyű kivétel, jelzi a számítógépnek, hogyan értelmezze a gépelést. Ezek közül a legfontosabb a *RETURN* billentyű. A *RETURN* jelzi a beírt sor végét, a rendszer úgy visszhangozza, hogy a kurzort (mutatót) a képernyő következő sorának elejére állítja. A rendszer a *RETURN* lenyomására értelmezi a begépelt karaktereket.

A *RETURN* vezérlőkarakter, egy láthatatlan karakter, amely valahogyan vezérli a terminál bemenetét és kimenetét. Minden komolyabb terminálon a *RETURN* önálló billentyű, a legtöbb vezérlőkarakter azonban nem, ezért azokat általában úgy kell bevinnünk, hogy lenyomva tartjuk a *CONTROL* billentyűt (szokásos rövidítései: *CTL, CNTL, CTRL*), és ugyanakkor lenyomunk egy másik billentyűt, általában egy betűt. Például a *RETURN*-t beírhatjuk a *RETURN* billentyű megnyomásával vagy a *CONTROL*-t lenyomva az *m* betűvel (Ezért a *RETURN*-t control-m-nek is hívjuk és *ctl-m*-nek jelöljük.) Egyéb vezérlőkarakterek pl.: a *ctl-d,* ami azt jelzi a programnak, hogy nincs több bemenet; a *ctl-g,* amely megszólaltatja a terminál csengőjét; a *ctl-h* a visszalépés (backspace), amit gépelési hibák javításakor alkalmazunk: a *ctl-i* a tabulátor, amely előrelépteti a kurzort a következő tabulátorstopig, mint a közönséges írógépen. Tabulátorstopok a UNIX rendszerekben alapértelmezés szerint 8 betűhelyenként vannak. A visszalépésnek és a tabulátornak a legtöbb terminálon külön billentyűje van.

Két további billentyűnek van különleges jelentése. A *DELETE* billentyűnek, amelyet néha *RUBOUT*-nak hívnak és a *BREAK* billentyűnek, amelyet *INTERRUPT*-nak is szoktak nevezni.

A UNIX rendszerek nagy részében a *DELETE* billentyű azonnal megállítja a

programot anélkül, hogy megvárná, amíg befejeződik. Bizonyos rendszerekben a *ctl-c* teszi ugyanezt. Néhány rendszerben a terminálok összekapcsolásától függően a *DELETE* vagy *ctl-c* szinonímája a *BREAK*.

A UNIX rendszer felügyelete alatt végzett munka menete

Kezdjünk egy magyarázatokkal fűszerezett párbeszédet a UNIX rendszerrel! Könyvünk példáiban
így jelezzük az általunk begépelt szöveget,
így a parancsokat és a fájlokat és
így a rendszer által kiírt üzeneteket.
A magyarázatokat pedig aláhúzással jelöljük.

Hozzuk létre a kapcsolatot: tárcsázzunk a telefonon, kapcsoljuk be a gépet, ahogy kell!
A rendszer a következőt kell, hogy mondja:

```
login: you          Írjuk be a nevünket, majd nyomjuk meg a RETURN-t!
Password:           A jelszót nem visszhangozza a rendszer.
You have mail.      Levelet kaptunk, olvassuk majd el!
$                   A rendszer várja parancsainkat.
$                   Nyomjuk meg néhányszor a RETURN-t!
$ date              Mi a dátum és időpont?
Sun Sep 25 23 : 02 : 57 EDT 1983.
$ who               Kik használják a gépet?
jlb       tty0        Sep 25  13 : 59
you       tty2        Sep 25  23 : 01
mary      tty4        Sep 25  19 : 03
doug      tty5        Sep 25  19 : 22
egb       tty7        Sep 25  17 : 17
bob       tty8        Sep 25  20 : 48
$ mail                        Elolvassuk a levelet.
From doug Sun Sep 25 20 : 53 EDT 1983
Hívj fel valamikor hétfőn
?                   RETURN hatására a következő levelünk jelenik meg.
From mary Sun Sep 25 19 : 07 EDT 1983. Következő levél.
Ebéd holnap délben?

? d                 Törli ezt a levelet.
$                   Nincs több levél.
$ mail mary         Levelet küldünk mary-nek.
Az ebéd 12-kor megfelel?
ctl-d               Levél vége.
$                   Lerakjuk a telefont vagy kikapcsoljuk a terminált és ezzel vége.
```

Néha ennyi egy munkamenet, bár előfordulhat, hogy valami munkát is el kell végeznünk. Az 1.1. szakasz további részében a fenti munkamenetet elemezzük, és más hasznos programokkal foglalkozunk.

Bejelentkezés

Mindenkinek rendelkeznie kell egy azonosítóval és egy jelszóval, amit a rendszerfelelős oszt ki.

A UNIX rendszer sokféle terminált képes kezelni, de erősen *kisbetű-orientált*; különbözőnek számít a kisbetű és a nagybetű. Csak nagybetűket író terminállal annyira bonyolult a munka, hogy okosabb egy másikat keresni. Győződjünk meg arról, hogy a kapcsolók beállítása megfelelő-e: kis- vagy nagybetű, full duplex és egyéb, a gépnél lévő szakemberek által ajánlott kapcsolók, pl. a sebesség- vagy baud rate! Létesítsünk kapcsolatot a rendszerrel a terminálunk típusától függő bűvös módon, ami lehet egy telefonhívás, lehet egy kapcsoló elfordítása. A rendszer válasza mindig

login:

kell, hogy legyen. Ha zagyvaságot ír, akkor pl. rossz lehet a sebesség: ezért ellenőrizzük a sebesség beállítását és a többi kapcsoló állását! Ha ez nem segít, nyomjuk meg néhányszor – lassan – a *BREAK* vagy az *INTERRUPT* billentyűt! Ha sehogy sem jelenik meg a bejelentkezési üzenet, segítséget kell kérnünk.

Ha megjelenik a **login:** írjuk be a nevünket *csupa kisbetűvel*, majd nyomjuk meg a *RETURN* billentyűt! Ha kell a jelszó, a gép megkérdezi, de beírása közben a képernyőre írás kikapcsolódik.

Bejelentkezési erőfeszítéseink megkoronázásaként egy *prompt* jelenik meg, ami rendszerint egyetlen karakter, ez jelzi, hogy a rendszer kész a parancsaink fogadására. A promt általában a $ (dollárjel), néha a % (százalék), de tetszés szerint bármire kicserélhetjük (kicsit később megmutatjuk, hogyan). Könyvünkben mi mindig a $ jelet fogjuk használni. A $ jelet a *burok* vagy *parancsértelmező* (shell) írja ki, ez a fő érintkezési felület köztünk és a rendszer között.

Lehet a $ előtt dátumkiírás vagy üzenet, hogy levelünk van. Kaphatunk arra vonatkozó kérdést is, hogy milyen típusú terminált használunk. Válaszunk segíti a rendszert abban, hogy a terminál esetleges sajátosságait kihasználja.

Parancsok beírása

Ha megkaptuk a $-t, akkor parancsokat írhatunk be, ezekkel kérhetjük a rendszertől, hogy csináljon valamit. A **$** után írjuk be a **date** parancsot, és nyomjuk meg a *RETURN*-t! A gép válaszként kiírja a dátumot és időpontot, majd újabb $-t, az egész folyamat valahogy így fog kinézni:

```
$ date
Mon Sep 26 12 : 20 : 57 EDT 1983
$
```

Ne felejtsük el a *RETURN*-t, és ne írjuk be a *$* jelet! Ha a rendszer nem figyel ránk, nyomjuk meg a *RETURN*-t, erre kell valaminek történnie. A *RETURN*-t a továbbiakban nem fogjuk mindig mondani, de ne felejtsük el minden sor végén megnyomni!

Most próbáljuk ki a **who** parancsot, ez megmondja hogy most éppen kik jelentkeztek be a rendszerbe!

```
$ who
rlm         tty0        Sep 26  11 : 17
pjw         tty4        Sep 26  11 : 30
gerard      tty7        Sep 26  10 : 27
mark        tty9        Sep 26  07 : 59
you         ttya        Sep 26  12 : 20
$
```

Az első oszlop a felhasználó neve, a második a rendszer által a létesített kapcsolatnak adott név (*tty* a *teletype* rövidítése, ugyanazt jelenti régiesen, mint ma a *terminál*).

A maradék a bejelentkezés időpontja. Próbálkozzunk meg a következővel:

```
$ who am i
you         ttya        Sep 26  12 : 20
$
```

Ha elhibáztuk a parancs nevének begépelését, és nem létező parancsra hivatkozunk, akkor a rendszer azt mondja, hogy ilyen nevű parancs nem található.

```
$ whom                  Hibás parancsnév... nem
whom: not found         találja a rendszer, nem tudja, hogyan futtassa.
$
```

Amennyiben véletlenül egy létező névre hibáztunk rá, futni fog, esetleg rejtélyes eredménnyel.

Furcsán viselkedő terminálok

Néha a terminál furcsa dolgokat művel, pl. minden karaktert kétszer ír vissza vagy a *RETURN* nem állítja a kurzort a következő sor első oszlopára. Ezek általában megszűnnek, ha a terminált kikapcsoljuk és újra bekapcsoljuk, vagy ha kijelentkezünk és újra bejelentkezünk, vagy ha elolvassuk a **stty** parancs (set terminal option) ismertetését a kézikönyv első részében. Ha a tabulátorok értelmes kezelését szeretnénk, de a terminálon nincs tabulátor, akkor írjuk be a

```
$ stty -tab
```

parancsot, és a rendszer a tabulátorokat megfelelő számú szóközzé konvertálja. Ha a terminálon vannak a számítógép által beállítható tabulátorpozíciók, akkor a **tabs** paranccsal be lehet állítani azokat megfelelően. (Ezt a parancsot

```
$ tabs terminaltipus
```

formában kell beírni – l. a **tabs** parancs leírását a kézikönyvben.)

Gépelési hibák

Ha gépelés közben hibáztunk és észrevesszük azt, mielőtt a *RETURN*-t lenyomjuk, kétféle módon javíthatunk: karaktereket egyenként *törölhetünk* vagy *kiirthatjuk* az egész sort és beírhatjuk újra. Ha a *sorirtó* karaktert, alapértelmezésben a @-t írjuk be, akkor az egész sor eltűnik, mintha sosem írtunk volna be, és új sorban kezdhetünk dolgozni:

```
$ ddtae@          Teljesen rossz, újra kezdjük egy új sorban.
date
Mon Sep 26 12.23.39 EDT 1983.
$
```

A # karakter törli az előtte álló karaktert, mindegyik # újabb karaktert töröl, visszafelé egészen a sor elejéig (azon túl már nem). A hibás gépelést a következő módon javíthatjuk:

```
$ dd#atte# #e
Mon  Sep  26  12.24.02 EDT 1983
$
```

A törlő- és sorirtó karakterek nagyon *rendszerfüggőek*. Sok rendszerben (a miénkben is) a törlőkarakter a backspace (visszalépés), ami *videoterminálokon* nagyon kellemesen működik. Gyorsan ellenőrizhető, hogy az adott rendszerben mi a helyzet:

```
$ datee←                    Próbáljuk a ←-t!
datee←: not found           Nem a←.
$ datee#                    Próbáljuk a #-t!
Mon Sep 26 12 : 26 : 08 EDT 1983.   Ez az!
$
```

(A visszalépést ← karakterrel jelöljük, hogy a szövegben látható legyen.) Másik gyakori választás a *ctl-u* mint sorirtó karakter. A továbbiakban a # karaktert használjuk törlésre, mert a szövegben jól látható, ha az adott rendszeren más van, akkor helyettesítsük azzal. Későbbiekben, a „Rendszerátalakítás" részben megmutatjuk, hogyan lehet a törlő- és sorirtó karaktereket megváltoztatni.

Mi történik akkor, ha a törlő- vagy sorirtó karaktert egy szöveg részeként be kell írnunk? Ha a # vagy @ előtt \ (fordított törtvonalat) írunk, akkor azok elveszítik különleges jelentésüket. Tehát, ha # vagy @ jelet kell bevinnünk, akkor \#, ill. \@ jeleket kell beírnunk. Lehet, hogy a rendszer a behívása után akkor is a következő sor elejére állítja a terminál kurzorát, ha írtunk előtte fordított törtvonalat. Ne aggódjunk, a jelet rögzítette a rendszer!

A fordított törtvonal, amelyet néha *escape* karakternek hívnak, általában azt jelzi, hogy a rákövetkező karakter valamiért különleges. A fordított törtvonalat két törlőkarakterrel lehet törölni: \##. (Érthető, hogy miért?)

A beírt karaktereket egy programsorozat ellenőrzi és értelmezi, mielőtt elérik rendeltetési helyüket. A pontos értelmezés nemcsak attól függ, hogy a karakterek hol végződnek, hanem attól is, hogy hogyan jutottak oda.

A beírt karakter azonnal visszhangzódik a terminálra (kivéve, ha a visszhangot kikapcsoljuk, amit ritkán teszünk meg). Amíg nem nyomunk *RETURN*-t, addig a mag átmenetileg tartja az adatokat, így a hibák a törlő- és sorirtó karakterekkel javíthatók. Ha fordított törtvonal előzi meg a törlő- vagy sorirtó karaktert, akkor a vezérlőprogram a fordított törtvonalat eldobja és a következő (törlő- vagy sorirtó) karaktert létező karakterként tárolja.

Amikor lenyomjuk a *RETURN*-t, akkor az összegyűjtött karaktereket a mag elküldi a terminálról olvasó programnak. Ez a program értelmezi a különleges karaktereket, pl. a burok törli a karakter különleges értelmezését, ha előtte fordított törtvonalat talált. A harmadik fejezetben erre még visszatérünk. Most csak annyit jegyezzünk meg, hogy a mag feldolgozza a törlő- és sorirtó karaktereket, továbbá a fordított törtvonalat akkor, ha törlő- vagy sorirtó karakter előtt áll, ami karakter megmarad, azt más programok is értelmezik.

1-1. feladat. Magyarázzuk el, mi történik:

```
$ date\@
```

□

1-2. feladat. A legtöbb burok (bár a 7. kiadásé nem) a # jelet magyarázat (comment) kezdetének értelmezi, a # után a sor végéig álló összes szöveget semmisnek tekinti. Ennek ismeretében magyarázzuk meg a következőket, feltételezve, hogy a törlőkarakterünk is a #.

```
$ date
Mon Sep 26 12 : 39 : 56 EDT 1983
$ date
Mon Sep 26 12 : 40 : 21 EDT 1983
$ \#date
$ \\#date
#date: not found
$
```

□

Előregépelés

A mag folyamatosan olvassa, amit beírunk, még akkor is, ha éppen mással van elfoglalva, tehát akkor és olyan gyorsan gépelhetünk, ahogy tetszik, az sem baj, ha közben valamilyen parancs nekünk nyomtat. Ha gépeltünk, miközben a rendszer ír, akkor a bemenő karaktereink keverednek a kimenő karakterekkel, de tárolásuk és értelmezésük megfelelő sorrendben történik. Folyamatosan gépelhetjük be a parancsokat anélkül, hogy befejeződésüket, sőt elindulásukat megvárnánk.

Programmegállítás

A parancsok nagy része leállítható a *DELETE* karakter beírásával. A legtöbb terminálon megtalálható *BREAK* általában ugyanezt csinálja, ez azonban rendszerfüggő. Néhány programban, pl. a szövegszerkesztőben a *DELETE* megszakítja a program tevékenységét, de benne maradunk a programban. A terminál kikapcsolása vagy a telefon lerakása is megszakítja legtöbb programot. Ha csak szünetet akarunk a kiírásban, pl. nem akarjuk, hogy valami fontos eltűnjön a képernyőről, nyomjuk meg a *ctl-s* billentyűket. A kiírás szinte azonnal megáll, programunk felfüggesztődik, míg újra nem indítjuk. Ha folytatni akarjuk, írjuk be *ctl-q* karaktert.

Kijelentkezés

A kijelentkezés megfelelő módja, hogy parancs helyett *ctl-d* karaktert írunk be, ez a burok számára azt jelenti, hogy nincs több bemenő jel. (Hogy ez valójában hogyan történik, azt a következő fejezetben elmagyarázzuk.) Általában elég, ha egyszerűen kikapcsoljuk a terminált vagy lerakjuk a telefont, de hogy ebből mikorra lesz igazi kijelentkezés, az a rendszertől függ.

Levelezés

A rendszer a felhasználók közötti érintkezést egy postarendszerrel támogatja, így előfordulhat, hogy bejelentkezés után az első **$** előtt a következő üzenet jelenik meg:

```
You have mail.
```

Hogy elolvashassuk a levelet, írjuk be ezt:

```
$ mail
```

A terminálra kiíródik a postánk, egyszerre csak egy levél, először a legfrissebb. A **mail** minden darab kiírása után várja, hogy megmondjuk, mit tegyen vele. A két alapválasz közül az egyik a **d,** ami törli az üzenetet, a másik a *RETURN,* ami nem törli (amikor újra megnézzük a postát, akkor is megtaláljuk ugyanezt a levelet). Egyéb lehetőségek: **p** újra kiírja a levelet, **s** *fájlnév* elteszi a megnevezett fájlba, **q** kilép a **mail** programból. (Aki nem tudja, mi a fájl, tekintse most egyszerűen olyan helynek, ahol információt tárolhatunk az általunk kiválasztott név alatt. A fájlok témájával az 1.2. szakaszban foglalkozunk, valójában azonban könyvünk nagy része erről szól.) A **mail** egyike azoknak a programoknak, amelyek valószínűleg nem egyeznek meg az itt leírtakkal, nagyon sok változatuk lehet. (A részleteket meg kell nézni a kézikönyvben.)

Levelet küldeni nagyon egyszerű. Ha a címzett **login** neve **nico,** akkor ennyi az egész:

```
$ mail nico
Most írjuk be a levél szövegét
akárhány sor lehet...
Az utolsó sor után nyomjuk meg a control-d-t!
ctl-d
$
```

A levél végét *ctl-d* jelzi, ez azt mondja, a **mail** programnak, hogy nincs több bemenő jel. Ha levélírás közben meggondoltuk magunkat, akkor *ctl-d* helyett nyomjunk *DELETE*-t. A félkész levél a **dead.letter** nevű fájlban tárolódik, nem kerül a címzetthez.

Gyakorlásként küldjünk levelet saját magunknak, majd írjuk be a **mail** parancsot és olvassuk el! (Nem olyan beteg dolog, mint amilyennek látszik – ügyes emlékeztető mechanizmus.)

Más üzenetközlő lehetőségek is vannak; elküldhetünk egy előre összeállított levelet, egyszerre több embernek elküldhetjük levelünket, és küldhetünk levelet más gépeken dolgozó embereknek. A részletek megtalálhatók a UNIX kézikönyv (UNIX Programmer's Manual) 1. fejezetében, a **mail** parancs leírásában. A továbbiakban az egyszerűség kedvéért a **mail (1)** jelöléssel fogunk a kézikönyv 1. fejezetére utalni. Az ebben a fejezetben ismertetett parancsok mind a kézikönyv 1. fejezetében találhatók.

A rendszerben lehet naptár is (l.: **calendar (1)**); a 4. fejezetben bemutatjuk, hogyan lehet naptárat csinálni, ha nincs a rendszerben.

Írás más felhasználóknak

Többfelhasználós UNIX rendszernél előfordulhat, hogy a terminál váratlanul csipog egyet és kiír valami ilyesmit:

```
Message from mary tty7 ...
```

mary szeretne írni nekünk, de ha nem cselekszünk valamit, akkor nem tudunk majd visszaírni. Hogy válaszolhassunk, írjuk be ezt:

```
$ write mary
```

Ezzel egy kétirányú érintkezési utat létesítettünk. Most mindaz, amit Mary leír az ő terminálján, az megjelenik a mi képernyőnkön és megfordítva, igaz hogy a csatorna olyan lassú, mintha a Holdhoz beszélnénk.

Ha éppen valaminek a közepén vagyunk, akkor olyan helyzetbe kell jutnunk, hogy beírhassunk egy parancsot. Általában bármilyen programot is futtatunk, azt meg kell állítani vagy meg kell állíttatni, de néhány program, mint a szövegszerkesztő és maga a **write,** rendelkezik egy ! paranccsal, amely ideiglenesen kimenekíti a burokba (l. a függelék F.2. táblázatát).

A **write** parancsnak nincsenek formai megkötései, ezért egyezséget kell kötni, különben az általunk beírtak összekeverednek azzal, amit Mary ír. Egy ilyen egyezség lehet az, hogy mindegyik szakasz végét *(o)*-val, a kilépési szándékot *(oo)*-val jelöljük.

Mary terminálja	A mi terminálunk
$ write you	
	$ Message from mary tty7 ...
	write mary
message from you ttya ...	
elfelejtetted az ebédet? (o)	*elfelejtetted az ebédet? (o)*
	öt @
	tíz perc (o)
tíz perc (o)	
ok (oo)	
	ok (oo)
	ctl-d
EOF	
ctl-d	
	$ EOF
$	

A **write**-ból a *DELETE*-tel is kiléphetünk. Figyeljük meg, hogy gépelési hibáink nem jelennek meg Mary terminálján!

Ha olyannak akarunk írni, aki nincs bejelentkezve, vagy aki nem akarja, hogy zavarják, azt a rendszer megmondja nekünk. Ha a címzett bejelentkezett ugyan, de nem válaszol egy ideig – lehet, hogy mással van elfoglalva, vagy elment a terminálról –, akkor írjuk be *ctl-d* vagy a *DELETE* karaktert. Ha nem akarjuk, hogy zavarjanak, használjuk a **mesg(1)** parancsot.

Újság

A UNIX rendszerek többségének van újság szolgáltatása, ezzel tájékoztathatók a felhasználók az érdekesebb vagy nem annyira érdekes eseményekről. Próbáljuk ki, és írjuk be a

$ *news*

parancsot! A UNIX rendszerek nagy hálózata telefonvonalakon keresztül van kapcsolatban egymással. Kérdezzük meg a helyi szakértőt a hálózati újságról és az **USENET**-ről. (Ez persze Magyarországra nem érvényes. *A szerk. megj.*)

A kézikönyv

A kézikönyv (UNIX Programmer's Manual) szinte mindent leír a rendszerről, amire a felhasználóknak szükségük lehet. Az 1. fejezet a parancsokkal foglalkozik, többek között azokkal, amiket mi itt tárgyalunk. A 2. fejezet a rendszerhívásokat ismerteti,

ez a mi 7. fejezetünk anyaga. A többi fejezet a C nyelven programozók által használható dolgokról, a fájlformátumokról és a rendszerkarbantartásról szól. (A fejezetek számozása rendszerenként változhat.) Használjuk a kézikönyv elején lévő tárgymutatót, ebben könnyen kikereshetjük azokat a parancsokat, amelyek kitűzött feladatunk megoldásánál hasznosak lehetnek. A bevezetésből áttekinthetjük ezek működését.

A kézikönyvet gyakran beírják a számítógép háttértárába, így azt terminálunkon megjeleníthetjük. Ha elakadunk és nincs szakértő segítségünk, akkor a kézikönyv bármelyik lapját kiírhatjuk terminálunkra a **man** *parancsnév* paranccsal.

Ha a **who** parancsról akarunk olvasni, írjuk be ezt:

```
$ man who
```

és persze a

```
$ man man
```

a **man** parancsról tájékoztat.

Számítógéppel segített tanulás

A rendszer tartalmazhat egy **learn** nevű parancsot, ami a fájlrendszer és az alapvető parancsok, a szövegszerkesztő, a dokumentumkészítő, sőt a C programozás számítógépes tanítója. Próbáljuk ki:

```
$ learn
```

Ha van **learn** a rendszerben, akkor ettől kezdve meg fogja mondani, hogy mit kell tenni. Ha hibádzik, megpróbálhatjuk a **teach** parancsot.

Játékok

Van, ahol tiltják, mégis az egyik legjobb módja annak, hogy a géppel, a terminállal megbarátkozzunk, ha játékokat játszunk. A UNIX rendszer a játékprogramok szerény választékával vásárolható, ezt azonban a legtöbb helyen kibővítik. Kérdezősködjünk, vagy nézzük meg a kézikönyv 6. fejezetét!

1.2. Mindennapos eszközök: fájlok, általános parancsok

A UNIX rendszer az információt fájlokban tárolja, ezek olyanok, mint a közönséges irodai akták. Minden fájlnak van neve, tartalma, tárolási helye, és tartozik hozzá néhány adminisztratív információ, pl. hogy ki a tulajdonosa és hogy mekkora. A fájl

tartalmazhat levelet, nevek és címek listáját, egy program forrásnyelvű utasításait, adatokat, amiket egy program használ, programokat végrehajtható formában, vagy egyéb nem szöveges anyagot.

A UNIX fájlrendszer szervezése olyan, hogy saját személyi tulajdonú fájljaink a többi felhasználó fájljaitól függetlenül kezelhetők, és a többieké sem zavarja a miénket. Számtalan fájlműveleteket végző program van, most csak a leggyakrabban használtakat tekintjük át. A 2. fejezetben rendszerezve tárgyaljuk a fájlrendszert, és bevezetünk további, fájlokkal kapcsolatos parancsokat. .

Fájlok létrehozása: a szövegszerkesztő

Ha egy cikket, levelet vagy programot akarunk írni, felmerül a kérdés, hogyan vigyük be az információt a gépbe? E feladatok nagy részét megoldhatjuk a *szövegszerkesztő* (*text editor*) nevű információtároló és -kezelő program segítségével. Szinte minden UNIX rendszerben van *képernyőszerkesztő* (*screen editor*) program is, ami kihasználva a modern terminálok előnyeit, a végrehajtott változtatásokat a képernyőn rögtön láthatóvá teszi. A két legnépszerűbb ilyen a **vi** és az **emacs.** Könyvünkben azonban egyik képernyőszerkesztőt sem ismertetjük, egyrészt nyomdatechnikai okokból, másrészt azért, mert nincs közöttük általánosan elterjedt változat.

Minden rendszerben megtalálható az **ed** nevű, régebbi szövegszerkesztő. Ez nem használja ki az egyes termináltípusok különleges tulajdonságainak előnyeit, ezért minden terminálon használható. Néhány más fontos program (közöttük némely képernyőszerkesztő) alapját is képezi, ezért mindenképpen érdemes megismerni. Pontos leírása az 1. függelékben található meg.

Bármelyik szövegszerkesztőt választjuk is, fájlok létrehozásához alaposan meg kell ismernünk azt. Itt az **ed**-et használjuk. Magyarázatunk így kötődhet valamihez, és biztosak lehetünk abban, hogy példáink futnak Olvasóink rendszereiben is. Egyébként mindenki azt a szövegszerkesztőt használhatja, amelyik a legjobban tetszik neki.

Ahhoz, hogy az **ed** segítségével létrehozzunk egy **vacak** nevű, valamilyen szöveget tartalmazó fájlt, a következőket kell tennünk:

```
$ ed                          Hívja a szövegszerkesztőt.
a                             Az ed szöveghozzáíró parancsa.
most írjunk be
bármilyen szöveget...
.                             Csak egy pontot írunk be, erre abbahagyja a szöveghozzáírást.
w vacak                       Kiírja a szöveget a vacak nevű fájlba.
37                            Az ed kiírja a fájlba kivitt karakterek számát.
q                             Kilépünk az ed-ből.
$
```

Az **a** parancs (append) azt mondja **ed**-nek, hogy kezdje gyűjteni a szöveget. A . a szöveg végét jelzi, ezt új sor elejére kell írni, más nem lehet a sorban. Ne felejtsük el, mert amíg ezt be nem írjuk, addig semmilyen más **ed** parancsot nem ismer fel a gép – mindent, amit beírunk, hozzáteendő szövegnek tekint!

A szövegszerkesztő **w** parancsa (write) a beírt információt az adott – **w vacak** parancs esetén a **vacak** – nevű fájlban tárolja. A fájlnév tetszőleges szó lehet, a **vacak** nevet azért választottuk, hogy jelezzük, fájlunk nem igazán jelentős.

Az **ed** kiírja, hogy hány karaktert tett a fájlba. Amíg a **w** parancsot ki nem adjuk, semmit sem tárol az adott nevű fájlban, ha tehát félbehagyjuk a munkát és hazamegyünk, akkor az információt nem tárolja a **vacak** nevű fájlban. (Ha szerkesztés közben hagyjuk abba a munkát, akkor adataink, amikkel dolgoztunk, kimentődnek egy **ed.hup** nevű fájlba, amelyet a következő munkamenetben folytathatunk.) Ha szerkesztés közben a rendszer összeomlik (valamilyen szoftver- vagy hardverhiba miatt leáll), akkor a fájlban csak az utolsó írás parancs előtti adatok lesznek meg. A **w** parancs hatására az információ az adott fájlban rögzítődik, ha később újra el akarjuk érni, írjuk be ezt:

```
$ ed vacak
```

Természetesen a beírt szöveget lehet szerkeszteni, a helyesírási hibákat javítani, szórendet cserélni, átrendezni a sorokat stb. Ha kész vagyunk a **q** (quit) paranccsal, elhagyhatjuk a szövegszerkesztőt.

Milyen fájljaink vannak?

Hozzunk létre két fájlt, a **vacak**-ot és a **múlandó**-t (Arany János fordításából):

```
$ ed
a
Lenni vagy nem lenni:
.
w vacak
23
q
$ ed
a
az itt a kérdés.
.
w múlandó
17
q
$
```

Az **ed** által kiírt karakterszámba a sorvégi újsor- (newline) karakter is beleszámít, ezzel jelzi a rendszer a *RETURN*-t.

Az **ls** parancs listázza a fájlok neveit (nem a tartalmukat):

```
$ ls
múlandó
vacak
$
```

ami valóban az előbb létrehozott két fájl. (Lehetnek köztük olyanok is, amiket nem mi magunk hoztunk létre.) A nevek automatikusan ábécésorrendbe rendezettek.

Az **ls** mint a parancsok többsége, *opciókkal* rendelkezik, amikkel megváltoztathatjuk az alapértelmezés szerinti viselkedését. Az opciók a parancsnevet követik a parancs sorában, és rendszerint egy – (mínusz-) jellel kezdődnek, amit egybetűs opciók követnek. Például az **ls –t** hatására a fájlokat idő (time) szerinti rendben listázza ki; minél később változtattunk rajta, annál előbb írja ki:

```
$ ls –t
múlandó
vacak
$
```

Az **–l** opció hosszú (long) listát ad, több információt közöl a fájlokról:

```
$ ls –l
total 2
–rw–r––r––  1 you   23 Sep 26 16 : 25 vacak
–rw–r––r––  1 you   17 Sep 26 16 : 26 múlandó
$
```

A total 2 megadja, hogy a fájlok hány blokknyi lemezterületet foglalnak el; egy blokk általában 512 vagy 1024 karakter. A –rw–r––r–– karaktersorozat azt mutatja, hogy ki az, aki írhatja is a fájlt. Jelen esetben a tulajdonos (*you*) írhatja és olvashatja, a többiek csak olvashatják. A rákövetkező 1 a láncszemek száma (*link*), felejtsük el ezt a 2. fejezetig! Az ezután álló you a fájl tulajdonosa, az a személy, aki létrehozta. 23 és 17 a fájlok karaktereinek száma, megegyezik azzal, amit az **ed**-től kaptunk. A dátum és az idő azt mutatja, hogy a fájl mikor változott utoljára.

Az opciók összekapcsolhatók: **ls –lt** ugyanazokat az adatokat írja ki, mint **ls –l**, az újabbakat előre szedve. A **–u** opció megadja, hogy mikor használtuk a fájlt: az **ls –lut** használat szerinti időrendbe szedett hosszú (**–l**) listát ad. Az **–r** opció megfordítja a rendezést, így **ls –rt** a legrégebben változtatottakat írja előre. Ha megnevezzük azt a fájlt, ami iránt érdeklődünk, akkor az **ls** csak arról ad információt:

```
$ ls –l vacak
–rw–r––r––  1 you   23 Sep 26 16 : 25 vacak
$
```

A parancs sorban a program nevét követő karaktersorozatokat, mint amilyen fenti példánkban a **–l** vagy a **vacak**, a program *argumentumainak* nevezzük. Az argumentumok általában opciók vagy a parancs által használandó fájlok nevei.

Az opciók szokás szerint mínuszjellel kezdődnek, és mindegyik opció egybetűs, például **–t** vagy **–lt**. Általában, ha egy parancs elfogad ilyen opcionális argumentumokat, akkor azok megelőzik a fájlnév argumentumokat, de egyébként bármilyen sorrendben állhatnak. A UNIX programok szeszélyesen kezelik a többszörös opciókat. Például a szabványos 7. kiadás **ls** parancsa nem fogadja el az **ls –lt** szinonímájaként a

formát, míg más programok *igénylik* a többszörös opciók szétválasztását.

Ahogy haladunk, látni fogjuk, hogy az opcionális argumentumoknál kevés szabályosság figyelhető meg. Minden parancsnak megvan a maga lelki élete, mindegyiknél mást jelentenek a betűk (gyakran ugyanazt a funkciót más parancs másképp jelöli). Ezt a zavaró kiszámíthatatlanságot gyakran a rendszer szégyenfoltjaként emlegetik. Bár a helyzet javul − az új változatokban egyre több az egységesség −, csak azt tanácsolhatjuk, hogy Olvasóink igyekezzenek jobb programokat írni, és a kézikönyvet mindig tartsák elérhető közelségben!

Fájlok kinyomtatása; cat és pr

Ha már vannak fájljaink, hogyan nézhetjük meg, mi van bennük? Sok program van erre, több is, mint amennyi kellene! Az egyik lehetőség, hogy a szövegszerkesztőt használjuk:

```
$ ed vacak
23                          Az ed jelenti, hogy 23 karakter van a vacak-ban.
1,$p                        Kiírja a sorokat az elsőtől az utolsóig.
Lenni vagy nem lenni:       A fájlnak csak egy sora van.
q                           Készen vagyunk.
$
```

Az **ed** azzal kezdi, hogy jelenti a **vacak** karaktereinek számát, majd az **1,$p** parancs hatására kiírja a fájl összes sorát. Ha megtanuljuk a szövegszerkesztő használatát, kiválaszthatjuk, hogy mely részeket írassunk ki.

Van olyan eset, amikor nem lehet a szövegszerkesztőt használni kiírásra. Például az **ed** nem tud néhány ezer sornál nagyobb fájlt kezelni. Ezenkívül egyszerre csak egy fájlt ír ki, nem lehet vele több fájlt egymás után, folyamatosan kiíratni. Lássunk tehát néhány más módszert!

Az első, a legegyszerűbb, a kiíratő parancs, a **cat***, amely az argumentumaiban megnevezett összes fájl *tartalmát* kiírja:

```
$ cat vacak
Lenni vagy nem lenni:
$ cat múlandó
az itt a kérdés.
$ cat vacak múlandó
Lenni vagy nem lenni:
az itt a kérdés.
$
```

* A **cat** parancs neve a catenate : összekapcsol, felfűz; ill. a concatenate : összekapcsol szavakból származik. *(A szerk.)*

A megnevezett fájl vagy fájlok a terminálra láncolódnak, tartalmuk a terminálra egymás után, szünet nélkül, folyamatosan íródik.

Rövid fájloknál nincs probléma, de a hosszúaknál, ha nagysebességű kapcsolatunk van a géppel, igencsak igyekeznünk kell, hogy a *ctl-s* nyomkodásával megállítsuk a kiírást, mielőtt eltűnik a képernyőről. Nincs „szabványos" parancs, amely a fájlból egyszerre mindig egy teljes képernyőnyit ír ki, de szinte minden UNIX rendszerben van valami ilyesmi. Olvasóink rendszerében lehet, hogy megvan a **pg** vagy a **more.** A miénken a **p** nevűt használjuk, a 6. fejezetben megmutatjuk, hogyan lehet implementálni.

A **cat**-hoz hasonlóan a **pr** is kiírja a fájl tartalmát, de sornyomtatóknak megfelelő formátumban: minden oldal 66 sor (11 hüvelyk) hosszú, minden lap tetejére felírja a fájl utolsó változásának dátumát és az időpontját, az oldalszámot, a fájlnevet, és a lapváltáshoz néhány sort átugrik. Nyomtassuk ki a **vacak**-ot, majd lapdobás után a **múlandó**-t:

```
$ pr vacak múlandó
Sep 26 16 : 25 1983 vacak Page 1
Lenni vagy nem lenni:
     (további 60 üres sor)
Sep 26 16 : 25 1983 múlandó Page 1
az itt a kérdés.
     (további 60 üres sor)
$
```

A **pr** nyomtathat többoszlopos formátumban is:

```
$ pr −3 fájlnevek
```

mindegyik fájlt háromoszlopos formátumban írja ki. Bármilyen ésszerű értéket tehetünk a **3** helyére, a **pr** megteszi, ami tőle telik. (A *fájlnevek* helyére fájlok neveit kell beírni.) A **pr −m** a fájlokat párhuzamosan oszlopokba írja, l. pr(1).

Meg kell jegyeznünk, hogy a **pr** nem formázóprogram, nem rendezi át a sorokat, nem igazítja a lapszéleket. Az igazi formázók az **nroff** és a **troff,** amiket a 9. fejezetben fogunk ismertetni.

Vannak olyan parancsok, amelyek nagysebességű nyomtatóra írják a fájlokat. Nézzük meg a kézikönyvben a **lp, lpr** neveket, vagy üssük fel a tárgymutatót a *printer* szónál! A gépünkhöz kapcsolt berendezésektől függően dönthetünk, hogy melyiket használjuk. A **pr** és **lpr** parancsokat gyakran együtt használjuk, a **pr** megfelelően formázza az információt, az **lpr** a sornyomtatóra juttatás mechanizmusát kezeli. Kicsit később erre még visszatérünk.

Fájlok mozgatása, másolása, törlése; mv, cp, rm

Nézzünk néhány újabb parancsot! Először változtassuk meg a fájl nevét. Az átnevezés úgy történik, hogy az egyik névből „átmozgatjuk" egy másikba, valahogy így:

```
$ mv vacak pontos
```

Ez annyit jelent, hogy az eddig **vacak**-nak hívott fájl neve ezentúl **pontos** lesz, tartalma változatlan marad. Az **ls** parancs most már listát ad, amelyről a **vacak** hiányzik, helyette ott a **pontos**:

```
$ ls
pontos
múlandó
$ cat vacak
cat: can't open vacak
$
```

Vigyázzunk, ha egy fájlt egy már létező másikba mozgatunk át, akkor a célba vett fájl felülíródik!

Ha *másolatot* szeretnénk a fájlról (két változatot szeretnénk megőrizni), használjuk a **cp** parancsot:

```
$ cp pontos pontos.save
```

amelynek eredményeként a **pontos** nevű fájlról lesz egy **pontos.save** nevű másodpéldányunk.

Végül, ha már meguntuk a dolgot, az **rm** paranccsal törölhetjük a megnevezett fájlokat:

```
$ rm vacak múlandó
rm: vacak nonexistent
$
```

Figyelmeztetést kapunk, a törlendő fájlok valamelyike nem létezik, egyébként az **rm**, miként a UNIX parancsok nagy része, szép csendben működik. Semmi irka-firkalocsogás, a hibaüzenetek kurták, és néha nem is segítenek. A rövidség zavaró lehet a kezdők számára, a tapasztalt felhasználók viszont a szószátyár parancsokat érzik bosszantónak.

Miből áll a fájlnév?

Eddig sok fájlnevet használtunk, de arról nem szóltunk, hogy mik lehetnek fájlnevek, ezért ideje néhány szabállyal megismerkednünk. Először: a fájlnevek legfeljebb 14 karakter hosszúak lehetnek. Másodszor: fájlnévként ugyan majdnem minden karakter megengedett, a józan ész azonban azt sugallja, hogy csak látható karaktereket használjunk, és kerüljük azokat, amelyeknek más jelentése is lehet. Például mint azt az **ls** parancsnál láttuk, **ls −t** idő szerinti rendben listáz. Így, ha egy fájlunk neve **−t**, igencsak el kell gondolkodnunk, hogyan listázhatjuk név szerint. (Hogyan is?) Az első karakterként használt mínuszjelen kívül vannak egyéb különleges jelentésű karakterek. A csapdákat úgy kerülhetjük el, hogy amíg nem ismerjük jól a helyzetet, csak betűket, számokat, pontot és aláhúzást használunk. (A pontot és az aláhúzást a

fájlnevek darabolására szoktuk használni, mint a fenti **pontos.save** példában.) Végül ne feledjük: a kis- és nagybetűk különbözőknek számítanak – **vacak, Vacak** és **VACAK** három különböző név.

Egy maroknyi hasznos parancs

Most, hogy már megismerkedtünk a fájlok létrehozásának, neveik listázásának, tartalmuk kiíratásának fortélyaival, lássunk néhány fájlkezelő parancsot. Hogy magyarázatunk kötődjék valamihez, egy **vers** nevű fájlt fogunk használni, amely Augustus De Morgan ismert versét fogja tartalmazni. Hozzuk létre az **ed**-del a fájlt:

```
$ ed
a
Nagy bolhák hátán kis bolhák:
   csípõs bájt lelni ebben.
S még kisebb bolhák megtoldják
   e sort, mely végtelen.
Nagy bolhák rendje még nagyobb
   bolhák rendjét elõzi:
S még nagyobbakét... tudhatod,
   jön még nagyobb, s a többi.
.
w vers
232
q
$
```

Az első új parancs, amely egy vagy több fájlban megszámlálja a sorokat, szavakat, karaktereket, a **wc** nevezetű (word-counting : szószámlálás):

```
$ wc vers
 8  36  232  vers
$
```

Ez azt jelenti, hogy a vers 8 sorból, 36 szóból és 232 karakterből áll. A *szó* definíciója nagyon egyszerű, egy olyan karaktersorozat, amelyben nincs szóköz-, tabulátor- vagy újsor-karakter.

A **wc** több fájlt is végig tud számlálni (a végösszegeket írja ki), a számlálók bármelyike elnyomható. (Lásd wc(1))!

A második parancs neve **grep**; ez fájlok soraiban minta szerinti egyezést keres. (A név az **ed** program g/*szabályos-kifejezés*/p parancsától származik, l. az 1. függeléket.) Nézzük meg, hol van „bolhák" a **vers**-ben:

```
$ grep bolhák vers
Nagy bolhák hátán kis bolhák:
S még kisebb bolhák megtoldják
```

```
Nagy bolhák rendje még nagyobb
  bolhák rendjét előzi;
$
```

A **grep** *program* **−v** opciójával a mintát nem tartalmazó sorokat is kereshetjük (a **v** név az **ed** megfelelő parancsára utal, tekintsük egyszerűen az egyezés megfordításának).

```
$ grep −v bolhák vers
  csípős bájt lelni ebben.
  e sort, mely véghetetlen.
S még nagyobbakét... tudhatod,
  jön még nagyobb, s a többi.
$
```

A **grep** paranccsal több fájlt is végignézhetünk, ilyenkor az egyező sorok elé kiírja a fájlnevet is, innen tudjuk, hol talált egyezést. Vannak számláló, sorszámozó stb. opciók is. A **grep** az egyszavas *bolhák*-nál sokkal bonyolultabb mintákat is tud kezelni, erről a 4. fejezetben lesz majd szó.

A harmadik parancs a **sort,** ami a bemenetet soronként, ábécé szerint rendezi. Egy vers esetében persze ennek nincs sok értelme, de azért nézzük, mit csinál:

```
$ sort vers
  bolhák rendjét előzi;
  csípős bájt lelni ebben.
  e sort, mely véghetetlen.
  jön még nagyobb, s a többi.
Nagy bolhák hátán kis bolhák:
Nagy bolhák rendje még nagyobb
S még kisebb bolhák megtoldják
S még nagyobbakét... tudhatod,
$
```

A rendezés soronként történik, de az alapértelmezés szerinti rendezés a szóközöket teszi előre, majd a nagybetűk, végül a kisbetűk következnek, tehát nem szigorúan ábécé szerint rendez. A **sort** számtalan opciójával nagyon sokféle rendezési szempontot adhatunk meg − például érték szerint csökkenő sorrendbe vagy szótár szerinti rendezést, vezető szóközöktől eltekintő, soron belüli mezők szerinti stb. rendezést kérhetünk. Az opciókat lehetetlen megjegyezni, meg kell nézni a kézikönyvben. Néhány gyakrabban használt opció:

sort −r	Csökkenő normál rendezés.
sort −n	Szám szerinti rendezés.
sort −nr	Szám szerinti csökkenő rendezés.
sort −f	Kis- és nagybetűk nem különbözőek.
sort +n	Rendezést az n+1-edik mezőnél kezdi.

A 4. fejezet ad erről részletesebb információt.

Újabb fájlvizsgáló parancs a **tail**, ami kiírja a fájl utolsó tíz sorát. Nyolcsoros költeményünk esetén ezt nem érzékelhetjük, de nagyobb fájloknál jól használható. A **tail** egyik opciójával megadhatjuk a kiírandó sorok számát, íme a **vers** utolsó sora:

```
$ tail -1 vers
  jön még nagyobb, s a többi.
$
```

A **tail** azt is tudja, hogy a fájl kiíratását egy megadott sornál kezdje:

```
$ tail +3  fájlnév
```

hatására a harmadik sornál kezdi a nyomtatást. (Figyeljük meg a szokás szerint mínuszjellel kezdődő argumentum természetes megfordítását!)

Az utolsó két parancs fájlok összehasonlítására szolgál. Legyen a **vers** egy változata az **új_vers** fájlban:

```
$ cat vers
Nagy bolhák hátán kis bolhák:
  csípős bájt lelni ebben.
S még kisebb bolhák megtoldják
  e sort, mely véghetetlen.
Nagy bolhák rendje még nagyobb
  bolhák rendjét előzi;
S még nagyobbakét... tudhatod,
  jön még nagyobb, s a többi.
$ cat új_vers
Nagy bolhák hátán kis bolhák:
  csípős bájt lelnek ebben.
S még kisebb bolhák megtoldják
  a sort, mely véghetetlen.
Nagy bolhák rendje még nagyobb
  bolhák rendjét előzi;
S még nagyobbakét... tudhatod,
  jön még nagyobb, s a többi.
$
```

A két fájl között nincs sok különbség, szemmel alig lehet észrevenni. Ilyenkor jönnek jól a fájl-összehasonlító parancsok. A **cmp** megkeresi az első olyan helyet, ahol a két fájl különbözik:

```
$ cmp vers új_vers
  vers új_vers differ: char 49, line 2
$
```

Ez az üzenet azt jelenti, hogy a fájlok második sorai különbözőek, ami igaz, de nem mondja meg, hogy miben különböznek, és a **cmp** nem keres tovább.

A másik összehasonlító parancs a **diff**, ami kiírja mindazokat a sorokat, amelyekben változás, hozzáadás törlés van:

```
$ diff vers új_vers
2c2
< csípős bájt lelni ebben,
---
> csípős bájt lelnek ebben.
4c4
< e sort, mely véghetetlen.
---
> a sort, mely véghetetlen.
$
```

Ez azt jelenti, hogy az első fájl (**vers**) 2. sora a második fájl (**új_vers**) 2. sorává változott, a 4. sorral ugyanez történt.

A **cmp** parancsot általában akkor használjuk, ha meg akarunk győződni arról, hogy a két fájl tartalma valóban azonos. Gyorsan működik, bármilyen fájlra alkalmazható, nem csak szövegre. A **diff** parancsot akkor használjuk, ha azt várjuk, hogy a fájlok valamiben eltérnek egymástól, és tudni szeretnénk, hogy pontosan mely sorokban különböznek. A **diff** csak szöveges fájlokon működik.

A fájlrendszer parancsainak áttekintése

Az 1.1. táblázatban röviden összefoglaljuk az eddig megismert, fájlokkal kapcsolatos parancsokat.

1.1. táblázat

Fájlrendszerparancsok

ls	kiírja az aktuális jegyzékben levő összes fájl nevét
ls *fájlnév*	csak a megnevezett fájlokat írja ki
ls −t	idő szerinti rendben listáz, először a legkésőbbit
ls −l	hosszú lista, több információval, **ls −lt** is
ls −u	használati idő szerinti rendben listáz, **ls −lu**, **ls −lut** is
ls −r	fordított sorrendben listáz, **−rt**, **−rlt** stb. is
ed *fájlnév*	a megnevezett fájlt szerkeszthetjük
cp *fájl1 fájl2*	*fájl1*-et *fájl2*-be másolja, felülírja a *fájl2*-t, ha létezik
mv *fájl1 fájl2*	átmozgatja *fájl1*-et *fájl2*-be, felülírja *fájl2*-t, ha létezik
rm *fájlnevek*	törli a megnevezett fájlokat, visszahozhatatlanul

cat *fájlnevek*	kiírja a megnevezett fájlok tartalmát
pr *fájlnevek*	kiírja a tartalmat fejléccel; oldalanként 66 sor
pr $-n$ *fájlnevek*	*n* oszlopban ír ki
pr $-m$ *fájlnevek*	egymás mellett írja ki a megnevezett fájlokat (több oszlopban)
wc *fájlnevek*	fájlonként megszámlálja a sorokat, szavakat, karaktereket
wc $-$**l** *fájlnevek*	fájlonként megszámlálja a sorokat
grep *minta fájlnevek*	kiírja a mintával egyező sorokat
grep $-$**v** *minta fájlok*	kiírja a mintával nem egyező sorokat
sort *fájlnevek*	soronként ábécé szerint rendezi a fájlokat
tail *fájlnév*	a fájl utolsó 10 sorát írja ki
tail $-n$ *fájlnév*	a fájl az utolsó *n* sorát írja ki
tail $+n$ *fájlnév*	a fájlt az *n*-edik sortól kezdi kiírni
cmp *fájl1 fájl2*	kiírja az első eltérés helyét
diff *fájl1 fájl2*	kiírja a fájlok közötti eltéréseket

1.3. Még valami a fájlokról: a jegyzékek

A rendszer megkülönbözteti a mi **vacak** nevű fájlunkat minden más felhasználó azonos nevű fájljától. Ezt úgy éri el, hogy a fájlokat *jegyzékekbe* (directories) csoportosítja, ami olyan, mint egy könyvtárban a könyvek katalógusrendszere. Így nem okoz ütközést, ha különböző jegyzékekben azonos nevek szerepelnek.

Általában minden felhasználónak van egy *saját jegyzéke* (home directory), ezt néha bejelentkezési jegyzéknek (login directory) is hívjuk, amely csak az őhozzá tartozó fájlokat tartalmazza. Bejelentkezéskor a saját jegyzékbe kerülünk. Munka közben váltogathatjuk, hogy melyik jegyzékben dolgozunk – ezt *munkajegyzéknek* (working directory) nevezzük – de a saját jegyzékünk marad változatlan. Ha semmi különöset sem csinálunk, akkor ha létrehoztunk egy új fájlt, az a munkajegyzékbe kerül. Miután belépéskor ez egyben a saját jegyzékünk is, ezért a fájl független a mások jegyzékében lévő azonos nevű fájloktól.

A jegyzék tartalmazhat más jegyzékeket és közönséges fájlokat (nagy jegyzékek hátán kis jegyzékek...). Ezt a szervezést jól szemléltethetjük a jegyzékek és fájlok fájával, amelyen a gyökértől elindulva, a megfelelő ágakon végighaladva a rendszerben lévő bármelyik fájlt meg lehet találni. Fordítva is lehet, akárhonnan elindulhatunk a gyökér felé.

Próbáljuk először az utóbbit! Használjuk a **pwd** parancsot (print working directory), amely kiírja annak a jegyzéknek a nevét, amelyben éppen vagyunk:

```
$ pwd
/usr/you
$
```

Eszerint most az **usr** jegyzékben levő *you* jegyzékben vagyunk, az **usr** pedig a *gyökérjegyzékben* (root directory) van, ez utóbbi neve rendszerint egyszerűen /. A / karakter a név elemeit választja szét; a korábban említett 14 karakteres határ a névelemekre értendő. Sok rendszerben a **/usr** jegyzék tartalmazza a rendszer normál felhasználóinak jegyzékeit. (A **pwd** akkor is valami ilyesmit ír ki, ha a saját jegyzékünk nem a **/usr/you,** így követhetjük az alábbiakat). Ha ezt írjuk be:

```
$ ls /usr/you
```

akkor pontosan ugyanazt a fájlnévlistát kapjuk, amit a sima egyszerű **ls** parancs adott. Ha nem adunk argumentumot, akkor az **ls** a munkajegyzék tartalmát írja ki, ha adunk jegyzéknevet, akkor a jegyzéknek a tartalmát listázza. Most próbáljuk meg ezt:

```
$ ls /usr
```

Ez egy hosszú névlistát ír ki, amelyben saját bejelentkezési jegyzékünket, a **you**-t is megtaláljuk.

A következő lépés, hogy megpróbáljuk magát a gyökeret kiíratni. Válaszként valami ilyesmit fogunk kapni:

```
$ ls /
  bin
  boot
  dev
  etc
  lib
  tmp
  unix
  usr
$
```

(Ne tévesszen meg senkit a / kettős jelentése: a gyökérnek is ez a neve, és ez választja el a fájlneveket is.) A felsoroltak nagy része szintén jegyzék, de például a **unix** fájl a UNIX magot tartalmazza végrehajtható formában. Bővebbet erről a 2. fejezetben találunk.

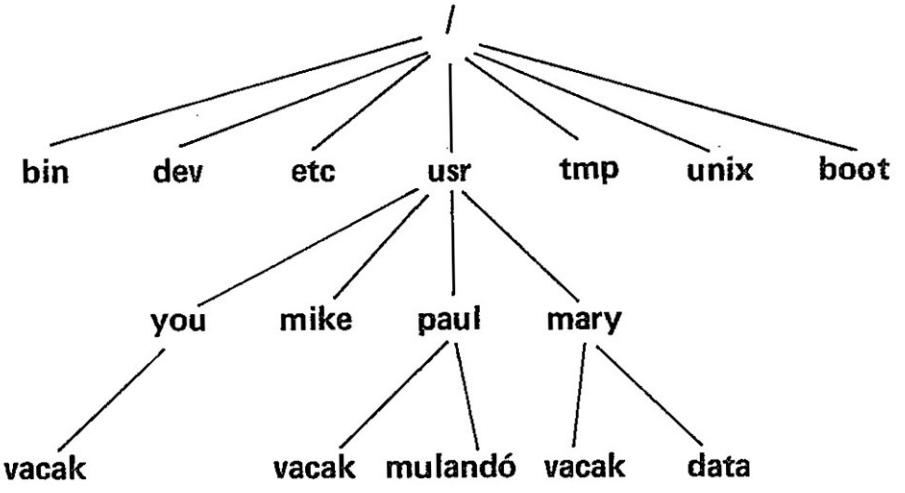
Most próbáljuk meg ezt:

```
$ cat /usr/you/vacak
```

(ha a **vacak** még mindig a jegyzékben van). A

```
/usr/you/vacak
```

a fájl ösvényneve („pathname"). Az „ösvénynév" intuitív jelentésű, a jegyzékek fáján a gyökértől az adott fájlig vezető ösvény teljes nevét jelenti. A UNIX rendszerben általános szabály, hogy mindenütt, ahol közönséges fájlnevet használhatunk, ott használhatjuk az ösvénynevet is.



A fájlrendszer szerkezete olyan, mint egy családfa; a rajz talán világosabbá teszi. A mi **vacak** nevű fájlunknak semmi köze sincs Paul vagy Mary **vacak**-jához.

Ha az összes fájl, amivel dolgozunk a saját jegyzékünkben van, akkor az ösvénynevek nem túl izgalmasak, de ha valakivel együtt vagy ha egyszerre több témán dolgozunk, akkor nagyon hasznosakká válnak. Például barátaink kinyomtathatják **vacak**-unkat, ha ezt mondják:

```
$ cat /usr/you/vacak
```

Hasonlóan megtudhatjuk, hogy Mary milyen fájlok tulajdonosa, ha ezt mondjuk:

```
$ ls /usr/mary
data
vacak
$
```

vagy készíthetünk magunknak másolatot az ő egyik fájljáról:

```
$ cp /usr/mary/data  data
```

vagy szerkeszthetjük az ő fájlját:

```
$ ed /usr/mary/data
```

Ha Mary nem akarja, hogy belekukucskáljunk a fájljaiba, a legteljesebb titoktartás is biztosítható. A hozzáférés ellenőrzésére minden fájlnak és katalógusnak van írási-olvasási-végrehajtási engedélye a tulajdonos, egy csoport és bárki más számára. A mi rendszereink felhasználói általában előnyösebbnek tartják a nyíltságot, mint a titkolódzást, de a körülmények Olvasóinknál mások lehetnek, erre a 2. fejezetben még visszatérünk.

Utolsó kísérletünk az ösvénynevekkel legyen ez:

```
$ ls /bin /usr/bin
```

Ismerősnek tűnik valamelyik név? Ha egy parancsot elindítunk úgy, hogy a $ után beírjuk a nevét, akkor a rendszer egy olyan nevű fájlt keres. Általában először a munkajegyzékben keresi (ahol valószínűleg nem találja), azután a **/bin**, végül a **/usr/bin** jegyzékben nézi meg. Nincs semmi különleges az olyan parancsokban, mint a **cat** vagy az **ls**, kivéve, hogy a könnyebb keresés és nyilvántartás érdekében össze vannak gyűjtve néhány jegyzékbe. Ezt bizonyítandó, próbáljunk e programok közül néhányat a teljes ösvénynevet használva végrehajtani:

```
$ /bin/date
Mon Sep 26      23 : 39 : 32 EDT 1983
$ /bin/who
srm      tty 1   Sep 26   22 : 20
cvw      tty 4   Sep 26   22 : 40
you      tty 5   Sep 26   23 : 04
```

1-3. feladat. Írjuk be ezt:

```
$ ls /usr/games
```

és válasszunk közülük! Lehetőleg munkaidő után! □

Jegyzékváltás; cd

Ha rendszeresen együtt dolgozunk Maryval az ő jegyzékében lévő információn, akkor mondhatjuk, hogy „Mary fájljain" akarunk dolgozni, nem a sajátjainkon. Ezt úgy tehetjük meg, hogy munkajegyzékünket a **cd** paranccsal megváltoztatjuk:

```
$ cd /usr/mary
```

Ha ezek után a **cat** vagy **pr** argumentumaként (/ jelek nélkül) használunk egy fájlnevet, akkor az a Mary jegyzékében levő fájlra vonatkozik. A jegyzékváltás nem változtatja meg a fájlhoz tartozó engedélyeket – ha egy fájlhoz saját jegyzékünkből nem tudunk hozzáférni, akkor más jegyzékre áttérve sem fogunk tudni.

Célszerű úgy elrendezni a saját fájljainkat, hogy az egy témával kapcsolatos fájlok külön jegyzékben, a többiektől elválasztva legyenek. Például, ha egy könyvet akarunk írni, az egész anyagot egy **book** nevű jegyzékben tarthatjuk. Új jegyzéket az **mkdir** paranccsal hozhatunk létre:

```
$ mkdir book        Létrehoz egy jegyzéket.
$ cd book           Belépünk.
$ pwd               Meggyőződünk arról, hogy jó helyen járunk-e.
/usr/you/book
...                 Írjuk a könyvet (ez sokáig eltart)!
$ cd ..             Egy szinttel feljebb lépünk a fájlrendszerben.
$ pwd
/usr/you
$
```

A (..) két pont az aktuális jegyzék, a munkajegyzék szülőjét, a gyökérhez egy szinttel közelebbi jegyzéket, az (.) egy pont pedig a munkajegyzéket jelenti. A saját jegyzékbe, amelybe a bejelentkezéskor kerültünk, az argumentum nélküli

```
$ cd                Saját jegyzékbe visszatérünk.
```

paranccsal térhetünk vissza.

Ha a könyv anyagára már nincs szükségünk, megszüntethetjük a fájlokat. A **book** jegyzéket csak akkor lehet törölni, ha előbb a benne lévő összes fájlt töröljük (hamarosan mutatunk egy gyors módszert). Ezután **cd**-vel a **book** szülőjegyzékbe megyünk, és beírjuk ezt:

```
$ rmdir book
```

Az **rmdir** csak üres jegyzéket töröl.

1.4. A burok

Amikor a rendszer kiírja a $-t és végrehajtandó parancsokat gépelünk be, akkor nem a maggal beszélgetünk, hanem egy közvetítővel, amelyet parancsértelmezőnek vagy *buroknak* (shell) nevezünk. A burok ugyanolyan közönséges program, mint a **date** vagy a **who,** bár figyelemre méltó dolgokat tud. Az a tény, hogy a burok köztünk és a mag lehetőségei között foglal helyet, számos előnyt rejt magában, néhányról fogunk beszélni. A három legfontosabb:

- Fájlnévrövidítések: programargumentumként egy egész sor fájlnevet felvehetünk azzal, hogy megadjuk a nevek mintáját. A burok ennek alapján megtalálja a minta szerint egyező fájlneveket.

• Kimenet – bemenet átirányítása: elintézhetjük, hogy valamely program kimenete a terminál helyett egy fájlra menjen, és hogy a bemenet a terminál helyett fájlról érkezzék. A bemenet – kimenet más programokhoz is kapcsolható.

• Személyesebbé tehetjük a rendszert: saját parancsokat és rövidítéseket adhatunk meg.

Fájlnévrövidítések

Kezdjük a fájlnévmintákkal! Tegyük fel, hogy egy nagy dokumentumot, mondjuk egy könyvet gépelünk. Ez logikailag sok kis részre tagozódik: fejezetekre, esetleg szakaszokra. Fizikailag is célszerű feldarabolni, mert nagy fájlokat nehéz szerkeszteni. Tehát a dokumentumot több fájlba gépeljük. Megtehetjük hogy minden fejezetet külön fájlba viszünk, mondjuk az **f1, f2** stb. nevűekbe. Amennyiben a fejezetek szakaszokra tagozódnak, fájljaink a következők lehetnek:

sz1.1
sz1.2
sz1.3
...
sz2.1
sz2.2
...

ahogyan azt ebben a könyvben is megfigyelhetjük. Ha rendszert viszünk a nevekbe, egy pillanat alatt megmondhatjuk, hogy melyik fájl hová tartozik.

Mit tegyünk, ha az egész könyvet ki akarjuk íratni? Mondhatjuk ezt

$ *pr sz1.1 sz1.2 ...*

de hamar unalmassá válik a sok fájlnév, és jönnek a gépelési hibák. Most lesz szükségünk a fájlnévrövidítésekre. Ha ezt mondjuk:

$ *pr sz**

a burok számára a * azt jelenti, hogy „bármely karaktersorozat", így az *sz** minta a munkajegyzék minden sz-szel kezdődő fájljánál egyezést ad. A burok összeállítja a listát ábécésorrendben* és átadja azt a **pr** programnak. A **pr** parancs sosem látja a * karaktert, ő a burok által a munkajegyzékben talált minta szerinti egyezések alapján kapott listát kapja.

* Itt sem szigorú ábécé szerint, a nagybetűk előbb jönnek, mint a kisbetűk. A rendezésben alkalmazott karaktersorrendet l. **ascii (7)**-ben.

A lényeg az, hogy a fájlnévrövidítés nem a **pr** parancs tulajdonsága, hanem a burok szolgáltatása, így ezzel bármelyik parancshoz készíthetünk fájlnévsorozatot. Például megszámlálhatjuk az első fejezetben a szavakat:

```
$ wc sz1.*
   113     562    3200  sz1.0
   935    4081   22435  sz1.1
   974    4191   22756  sz1.2
   378    1561    8481  sz1.3
  1293    5298   28841  sz1.4
    33     194    1190  sz1.5
    75     323    2030  sz1.6
  3801   16210   88933  total
$
```

Van egy **echo** nevű program, ami különösen alkalmas a fájlnévrövidítésekkel való kísérletezésre. Mint nevéből sejthető, az **echo** nem csinál mást, csupán visszhangozza argumentumait:

```
$ echo jó reggelt
jó reggelt
$
```

Az argumentumok azonban minta szerinti egyezéssel is előállíthatók:

```
$ echo  sz1.*
```

kiírja az 1. fejezet összes fájlnevét;

```
$ echo *
```

pedig ábécébe szedve kiírja a munkajegyzék *összes* fájlnevét;

```
$ pr *
```

kinyomtatja az összes fájl tartalmát (ábécésorrendben), és

```
$ rm *
```

törli a munkajegyzékben levő összes fájlt. (Nagyon vigyázzunk vele, valóban mindent töröl!).

A * nemcsak a fájlnév végén állhat, bárhol és többször is előfordulhat. Így a

```
$ rm *.save
```

törli az összes fájlt, ami **.save**-re végződik. Figyeljük meg, hogy a fájlnevek ábécébe szedve íródnak ki, ami nem ugyanaz, mint a szám szerinti rendezés, például ha a könyvnek 10 fejezete van, akkor a sorrend szándékunktól különbözhet, hiszen **sz10** előbb van, mint **sz2.**

```
$ echo *

sz1.1 sz1.2... sz10.1 sz10.2 ... sz2.1 sz2.2
$
```

A buroknak más mintakereső tulajdonságai is vannak, bár leggyakrabban a *-ot használjuk. A *[...]* minta a szögletes zárójeleken belüli bármely karakter szerinti egyezést keres. Az egymás utáni betűk vagy számjegyek rövidíthetők:

*$ pr sz [12346789] **	Kiírja az 1,2,3,4,6,7,8,9 fejezetet, az 5.-et nem.
*$ pr sz [1−46−9] **	Ugyanezt teszi.
$ rm temp[a−z]	Törli az összes létező **tempa**, ..., **tempz** fájlt.

A *?* minta egyetlen karaktert egyeztet:

$ ls ?	Kiírja az összes egykarakteres fájlnevet.
$ ls −l sz?.1	Az **sz1.1 sz2.1 sz3.1** stb. fájlokat listázza, a **sz10.1** fájlt nem.
$ rm temp?	Törli a **temp1**, ..., **tempa** stb. fájlokat.

Megjegyezzük, hogy az egyezés keresése csak létező nevekre működik, mintákkal nem lehet új fájlneveket létrehozni. Például, ha az sz helyett minden fájlnévben **szakasz**-t szeretnénk írni, azt így nem csinálhatjuk:

$ mv sz. szakasz.**	Nem működik!

mert a *szakasz.** nem létező fájlnevekkel mutat egyezést.

A mintakarakterek − mint a * − az ösvénynevekben ugyanúgy használhatók, mint az egyszerű fájlnevekben; az egyezést az ösvény mindegyik, a megadott karaktert tartalmazó elemében keresi. Így a */usr/mary/** a keresést a **/usr/mary** alatt végzi, és a */usr/*/naptár* az összes felhasználói **naptár** fájl ösvénynevének listáját adja.

Ha valamiért fel kell függesztenünk a *, ?* stb. különleges jelentését, akkor tegyük az argumentumot aposztrófok közé:

```
ls '?'
```

A különleges karaktereket a \ (fordított törtvonallal) is hatástalaníthatjuk:

```
$ ls \?
```

(Emlékezzünk vissza, hogy miután a ? nem törlő- vagy sorirtó karakter, a fordított törtvonalat nem a mag, hanem a burok kezeli!)

1-4. feladat. Mi a különbség az alábbi parancsok között?

$ *ls vacak*	$ *echo vacak*
$ *ls /*	$ *echo /*
$ *ls*	$ *echo*
$ *ls **	$ *echo **
$ *ls '*'*	$ *echo '*'*

□

A kimenet – bemenet átirányítása

Az eddig tárgyalt parancsok nagy része valamilyen kimenetet ad a terminálon; egyes parancsok, mint az **ed**, bemenetüket is a terminálról veszik. A terminál helyére majdnem mindig tehetünk kimenetként vagy bemenetként, vagy mindkettőként fájlokat. Például:

```
$ ls
```

fájlneveket, a terminálra listázza,

```
$ ls > fájllista
```

a fájlnevek előbbi listáját a terminál helyett a **fájllista** nevű fájlba írja. A > jelentése: „a kimenetet ne a terminálra, hanem a következő fájlba tedd". Ha nem létezik a fájl, akkor létrehozza, ha létezik, akkor előző tartalmát felülírja. A terminálra semmi sem kerül. Egy másik példa: több fájlból egyet csinálhatunk, ha a **cat** kimenetét egy fájlba tereljük:

```
$ cat f1 f2 f3 > múlandó
```

A ≫ jel hasonlóan működik, mint >, de nem töröl, hanem hozzáír a meglevő végéhez, vagyis

```
$ cat f1 f2 f3≫múlandó
```

a **múlandó**-hoz a meglevő tartalom folytatásaként hozzáírja **f1**, **f2** és **f3** tartalmát. Ugyanúgy, mint >, ha nincs **múlandó** nevű fájl, akkor üres fájlként létrehozza.

Hasonlóan, a < jel azt jelenti, hogy a program bemenete a terminál helyett legyen a következő fájl. Így például a **levél** fájlban összeállított levelet így küldhetjük el több embernek:

```
$ mail mary joe tom bob <levél
```

Példáinkban a > ill. < jelek egyik oldalán sem kell szóköz, azonban ez a szokásos formátum.

Miután az adott kimenet a > jellel átirányítható, olyan parancskombinációkra nyílik lehetőségünk, amelyek másképp nem elérhető eredményeket adnak. Például a felhasználók neveit kiírhatjuk ábécébe rendezve:

```
$ who >múlandó
$ sort <múlandó
```

Mivel a **who** soronként kiírja a bejelentkezett felhasználókat és a **wc −l** (a szó- és karakterszámlálást elnyomva) megszámlálja a sorokat, így számlálhatjuk meg, hogy hány felhasználó van bejelentkezve:

```
$ who >múlandó
$ wc −l <múlandó
```

A munkajegyzékben levő fájlokat így számlálhatjuk meg:

```
$ ls >múlandó
$ wc −l <múlandó
```

bár ez magát a **múlandó** fájlnevet is hozzászámolja. Kiírhatjuk a fájlneveket három oszlopba:

```
$ ls >múlandó
$ pr −3 <múlandó
```

A **who** és a **grep** összekapcsolásával megnézhetjük, hogy egy adott felhasználó bejelentkezett-e:

```
$ who >múlandó
$ grep mary <múlandó
```

Fontos megjegyezni, hogy ezekben a példákban a > és < értelmezését ugyanúgy, mint a * és más mintakereső karakterekét, nem az egyes programok, hanem a burok végzi. Ezáltal a bemenet és kimenet átirányítása bármelyik programnál alkalmazható; maga a program észre sem veszi, hogy valami szokatlan történt.

Ez felvet egy nagyon lényeges dolgot, a

```
$ sort <múlandó
```

a **múlandó** fájl tartalmát rendezi ugyanúgy, mint a

```
$ sort múlandó
```

mégis van különbség. Miután a **múlandó** karaktereket a burok értelmezi, a **sort** argumentumként nem is látja a **múlandó** fájlnevet, ő a szabványos bemenetet rendezi (amit a burok átirányított, ezért fájlból érkezik). A második példában viszont a **sort** argumentuma a **múlandó** név, a **sort** beolvassa a fájlt és rendezi azt. A **sort**-nak egy fájlnévlistát is adhatunk:

```
$ sort mul1 mul2 mul3
```

de ha nincs fájlnév, akkor a szabványos bemenetet rendezi. A legtöbb parancs lényeges tulajdonsága, hogy fájlnév hiányában a szabványos bemenetet dolgozza fel. Ez azt jelenti, hogy a parancsokat egyszerűen begépelve meglátjuk, hogyan működnek. Például:

```
$ sort
ghi
abc
def
ctl-d
abc
def
ghi
$
```

A következő szakaszban látni fogjuk, hogyan hasznosítható ez az elv.

1-5. feladat. Magyarázzuk meg, hogy

```
$ ls >ls.out
```

névlistájában miért szerepel *ls.out*! □

1-6. feladat. Mi lesz a

```
$ wc múlandó > múlandó
```

kimenete? Mi történik, ha elírjuk a parancs nevét, mint itt:

```
$ woh >múlandó
```

□

Csövek (pipes)

Az előző szakasz utolsó példái azonos trükköt használnak: az egyik program kimenetét egy ideiglenes fájlon keresztül egy másiknak a bemenetére adja. Az ideiglenes fájl másra nem is kell; valójában jobban járnánk, ha nem is volna. Ez a megfigyelés vezetett a UNIX rendszer egyik alapvető újításához, a *cső* (pipe) fogalmához. A cső

az egyik program kimenetét ideiglenes fájl nélkül kapcsolja egy másik bemenetére. A *csőhálózat* (pipeline) két vagy több program csöveken keresztüli kapcsolata.

Revideáljuk néhány korábbi példánkat, és használjunk átmeneti fájlok helyett csöveket. A | (2 függőleges vonal) karakter mondja meg a buroknak, hogy építsen csőhálózatot.

`$ who | sort`	Kiírja a felhasználók rendezett listáját.
`$ who | wc -l`	Megszámlálja a felhasználókat.
`$ ls | wc -l`	Megszámlálja a fájlokat.
`$ ls | pr -3`	Háromoszlopos fájlnévlista.
`$ who | grep mary`	Egy meghatározott felhasználót keres.

Minden program, amely a terminálról olvas, tud ahelyett csőből is olvasni. Minden program, amely ír a terminálra, tud csőbe is írni. Itt kamatozik az a megállapodás, hogy ha nincs fájlnév, akkor a szabványos bemenetet olvassa a rendszer: minden program, amely így működik, használható csőhálózatban. Így használtuk a **grep**, **pr**, **sort** és **wc** programokat a fenti csőhálózatokban.

Egy csőhálózatban akárhány programot összekapcsolhatunk:

```
$ ls | pr -3 | lpr
```

háromoszlopos fájlnévlistát ad a sornyomtatóra;

```
$ who | grep mary | wc -l
```

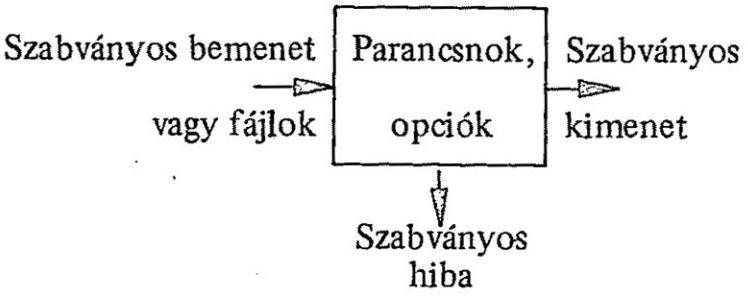
megszámolja, hogy Mary hány terminálon van bejelentkezve. Egy csőhálózat programjai valójában egy időben futnak, nem pedig egymás után. Ez annyit jelent, hogy a csőhálózat programjai interaktívak lehetnek, az egész dolog működéséhez szükséges ütemezésről, szinkronizálásról a mag gondoskodik.

Amint sejthetjük, a cső szervezését a burok végzi, maguknak a programoknak mindegy, hogy van-e átirányítás. A programoknak természetesen érzékenyen kell működniük, ha ily módon kívánjuk összerakni őket. A parancsok felépítése egységes, így bárhol pontosan illeszkednek a csőhálózatokra. A parancshívás általában ilyen formájú:

parancs opcionális_argumentumok opcionális_fájlnevek

Ha nincs megadva fájlnév, akkor a parancs a szabványos bemenetről olvas, amely alapértelmezésben a terminál (próbálgatásokhoz ez nagyon ügyes dolog), de át lehet irányítani úgy, hogy egy fájból vagy egy csőből érkezzen. Hasonlóan, a parancsok nagy része a szabványos kimenetre ír, amely alapértelmezésben a terminálra kerül, de átirányítható egy fájlba vagy egy csőbe.

A parancsok hibaüzeneteit másként kell kezelni, hiszen azok eltűnnének a fájlban vagy a csőben. Ezért minden parancsnak van egy *szabványos hibakimenete* is, ami alapértelmezésben a terminálra kerül. Az egész dolog felépítését ábránk mutatja:



Az eddig tárgyalt parancsok szinte mindegyike ilyen; kivéve az olyan parancsok, mint a **date** és a **who**, amelyek nem olvasnak a bemenetről és az olyanok, mint a **cmp** és a **diff**, amelyeknek rögzített számú fájl a bemenete. (A − opciót azonban azoknál is meg kell nézni).

1-7. feladat. Magyarázzuk meg, mi a különbség a

```
$ who | sort
```

és a

```
$ who >sort
```

□

között!

Folyamatok

A burok a csövek felépítésén kívül még sok egyéb dolgot csinál. Miután a csövekkel kapcsolatban már bepillantást nyertünk abba, hogy több programot egyidejűleg futtathatunk, nézzük meg most röviden ennek alapjait! Például futtathatunk két programot egy parancssorral, ha a parancsokat pontosvesszővel választjuk el; a burok felismeri a pontosvesszőt és a sort két parancsra darabolja:

```
$ date; who
Tue Sep 27 01 : 03 : 17 EDT 1983
ken     tty0  Sep 27  00 : 43
dm      tty1  Sep 26  23 : 45
rob     tty2  Sep 26  23 : 59
bwk     tty3  Sep 27  00 : 06
jj      tty4  Sep 26  23 : 31
you     tty5  Sep 26  23 : 04
ber     tty7  Sep 26  23 : 34
$
```

Mindkét parancsot végrehajtja (egymás után), ezután tér vissza a burok a $ karakterrel.

Ha tetszik, egyidejűleg több programot is futtathatunk. Például, tegyük fel, hogy valamilyen időigényes dolgot akarunk csinálni, mondjuk megszámlálni, hány szó van könyvünkben, de szeretnénk mást is csinálni, mielőtt a **wc** befejeződik. Ekkor a következőt mondhatjuk:

```
$ wc sz* >wc.out &
6944          A burok kiírja a folyamatazonosítót.
$
```

A parancssor végén álló & jel azt mondja a buroknak, hogy „kezdd el futtatni a parancsot, és rögtön vegyél újabb parancsokat a terminálról", vagyis ne várd meg, amíg befejeződik! Így a parancs elindul és amíg fut, mást csinálhatunk. A kimenetet a **wc.out** fájlra irányítottuk, hogy ne keveredjék össze azzal, amit közben csinálunk.

A futó program jelenségét *folyamatnak* (process) hívjuk. Azt a számot, amelyet a parancs & jellel való indításakor a burok kiír, *folyamatazonosítónak* (process-id) nevezzük, más parancsokban ezzel hivatkozhatunk egy adott futó programra.

Lényeges, hogy különbséget tegyünk a programok és a folyamatok között. Például **wc** egy program, valahányszor elindítjuk, annyiszor új folyamatot indítunk. Ha ugyanaz a program egyidejűleg több példányban fut, akkor mindegyik egy különálló folyamat, különböző folyamatazonosítóval.

Ha egy csőhálózatot & jellel indítunk, mint itt:

```
$ pr sz* | lpr &
6951                lpr egy folyamat azonosítója.
$
```

akkor annak folyamatai mind egyszerre indulnak, az & jel az egész csőhálózatra vonatkozik. Mégis csak egy folyamat azonosítója kerül kiírásra, sorrendben a legutolsóé. A

```
$ wait
```

parancs megvárja, amíg az & jellel indított összes folyamat befejeződik. Ha nem tér vissza rögtön, akkor van még futó parancsunk. A **wait** parancsot a *DELETE*-tel megszakíthatjuk.

A burok által kiírt folyamatazonosító felhasználásával megállíthatjuk az & jellel elindított folyamatot:

```
$ kill 6944
```

Ha elfelejtettük a folyamatazonosítót, a **ps** parancs megmondja, milyen folyamataink futnak. Ha kétségbeesünk, a **kill 0** paranccsal a bejelentkezési burok kivételével az összes folyamatunkat törölhetjük. Ha kíváncsiak vagyunk, mit csinálnak a többiek, a *ps −ag* az összes futó folyamatról tájékoztat, íme egy ilyen kiíratás:

```
$ ps −ag
PID  TTY TIME CMD
  36 co  6 : 29 (ltc) cron
6423 5   0 : 02 −sh
6704 1   0 : 04 −sh
6722 1   0 : 12 vi paper
4430 2   0 : 03 −sh
```

```
6612 7    0 : 03 −sh
6628 7    1 : 13 rogue
6843 2    0 : 02 write dmr
6949 4    0 : 01 login bimmler
6952 5    0 : 08 pr sz1.1 sz1.2 sz1.3 sz1.4
6951 5    0 : 03 lpr
6969 5    0 : 02 ps −ag
6844 1    0 : 02 write rob
$
```

ahol PID a folyamatazonosító, TTY a folyamathoz tartozó terminál (mint a **who** parancsban), TIME a felhasznált processzoridő percben, másodpercben, a többi a futó parancs neve. A **ps** azon kevés parancsok egyike, amelyek a rendszer egyes változataiban különbözőek, így lehet, hogy Olvasóink gépe másképp formázva írja ki ezeket az információkat. Még az argumentumok is lehetnek mások (l. a kézikönyv ps(1) lapját).

A folyamatoknak ugyanolyan hierarchikus szerkezetük van, mint a fájloknak: minden folyamatnak van szülője és lehetnek gyermekei. A bejelentkezési burkot az a folyamat hozza létre, amely a rendszerhez kapcsolódó terminálvonalakat figyeli.

Ha parancsokat indítunk, akkor a keletkező folyamatok bejelentkezési burkunk elsőszülött gyermekei. Ha ezek valamelyikétől indítunk egy programot, például az **ed**-ből kilépünk a ! paranccsal, akkor ez létrehozza a saját gyermek folyamatát, ami így a burok unokája lesz.

Néha olyan sokáig fut egy folyamat, hogy azt szeretnénk, ha az elindítás után kikapcsolhatnánk a terminált, s hazamehetnénk anélkül, hogy megvárnánk a végét. De ha kikapcsoljuk a terminált vagy megszakítjuk a kapcsolatot, a folyamat törlődik, még akkor is, ha a &-del indítottuk. Erre a helyzetre találták ki a **nohup** (no hangup) parancsot:

```
$ nohup parancs &
```

után a program a kijelentkezés ellenére is tovább fog futni. A parancs kimenete egy **nohup.out** nevű fájlban megőrződik. Visszaható parancsra nem lehet **nohup**-ot kérni!

Ha folyamatunk sok erőforrást köt le, akkor illik azt a szokásosnál alacsonyabb prioritással futtatni; ezt a **nice** nevű programmal tehetjük meg:

```
$ nice költséges_munka &
```

A **nohup** parancs automatikusan hívja a **nice**-t, mert ha úgyis kijelentkezünk, akkor megengedhetjük, hogy a parancs egy kicsit tovább tartson.

Végül megkérhetjük a rendszert, hogy valamilyen kevésbé leterhelt hajnali órán indítsa folyamatunkat, amikor a normális emberek nem számítógépeznek, hanem alszanak. Ez a parancs az **at(1)**:

```
$ at időpont
tetszés szerinti parancsok
ctl-d
$
```

Rendszerint így használjuk, de jöhetnek a parancsok egy fájlból is:

```
$ at 3am <file
$
```

24 órás vagy 12 órás formában adhatjuk meg (2130 vagy 930 pm).

A környezet igazítása

A UNIX rendszer egyik erénye, hogy számos lehetőséget ad arra, hogy egyéni ízlésünkhöz, a helyi számítástechnikai környezethez igazítsuk. Például már korábban említettük a különféle törlő- és sorirtó karakterek problémáját. Alapértelmezésben ezek a # és @ karakterek. Ezt az **stty** paranccsal bármikor megváltoztathatjuk:

```
$ stty erase e kill k
```

után az *e* lesz a törlő- és *k* a sorirtó karakter. Bosszantó azonban, hogy ezt a változtatást minden bejelentkezéskor meg kell tenni.

Újra a burok segít. Ha bejelentkezési jegyzékünkben van egy **.profile** nevű fájlunk, akkor a burok a bejelentkezéskor, az első $ kiírása előtt végrehajtja az ebben levő parancsokat. Így a **.profile** nevű fájlba tett parancsokkal tetszés szerinti környezetet alakíthatunk ki, ezek a parancsok minden bejelentkezéskor végrehajtódnak.

A legtöbben először azt írják a **.profile** nevű fájlukba:

```
stty erase ←
```

A ← jelet azért használjuk, hogy láthatóvá tegyük a visszalépés (backspace) karaktert, amit betettünk a **.profile** fájlba. A **stty** a *ctl-x* helyett a ^*x* jelölést is érti, úgyhogy ugyanazt a hatást az

```
stty '^h'
```

paranccsal is elérhetjük, hiszen *ctl-h a visszalépés.* (*A* ^ karakter a | csőoperátor szinonimája, ezért kell aposztrófok közé tenni.)

Ha terminálunkon nincsenek tabulátorstopok, a *-tabs* opciót is beírhatjuk az *stty* sorba:

```
stty erase '^h' -tabs
```

Ha szeretnénk tudni, hogy bejelentkezésünkkor mennyire foglalt a rendszer, tegyük még hozzá ezt:

```
who | wc -l
```

ami megszámlálja a felhasználókat. Ha van újságrendszer, akkor beírhatjuk a **news** parancsot is.

Egy idő után, ha már úgy érezzük, hogy túl sokáig tart a bejelentkezés, nekiláthatunk a **.profile** fájlunkba felvitt felesleges dolgok törlésének.

A burok egynémely tulajdonságait az ún. burokváltozókkal vezéreljük, ezeknek az értékeit elérhetjük és magunk is beállíthatjuk. Például a promptjel karakterlánca, amit **$** jellel jelölünk, valójában a **PS1** nevű burokváltozóban található, aminek tartalmát tetszés szerint átírhatjuk, akár így is:

```
PS1 = 'Ugye, kedvesem?'
```

A fél idézőjelek azért kellenek, mert a promptjel szövegében szóközök vannak, és ebben a szerkezetben az egyenlőségjel körül nem lehetnek szóközök.

A burok a **HOME** és **MAIL** változókat is speciálisan kezeli. A **HOME** a saját jegyzék neve, az normális esetben megfelelően beállítódik anélkül, hogy a **.profile** fájlban megadnánk. A **MAIL** változóban van annak a fájlnak a neve, amely postánkat tárolja.* Ha megadjuk a buroknak, akkor ha levelet kapunk, minden parancs után figyelmeztet erre:

```
MAIL = /usr/spod/mail/you
```

(A postafájl Olvasóink rendszerében lehet más is, pl. szokásos még **/usr/mail/you** is.)

Valószínűleg a leghasznosabb burokváltozó az, amelyik ellenőrzi, hogy a burok hol keresse a parancsot. Emlékezzünk vissza, hogy amikor beírjuk a parancs nevét, a burok először a munkakatalógusban, azután a **/bin**, végül a **/usr/bin** jegyzékben keresi azt. A jegyzéknek ezt a sorrendjét *keresési útnak* (search path) nevezik, és azt a **PATH** nevű burokváltozó tárolja. Ha az alapértelmezés szerinti keresési út nem kedvünkre való, akkor megváltoztathatjuk. Ezt a kívánságunkat is beírhatjuk a **.profile** fájlba. Például ez a sor a szabványos keresési úthoz hozzáteszi a **/usr/games** jegyzéket:

```
PATH =.:/bin:/usr/bin:/usr/games.   Egy lehetőség...
```

A szintaktika egy kicsit szokatlan: jegyzéknév-sorozat kettőspontokkal elválasztva. Emlékezzünk vissza, hogy a . a munkajegyzék. A . elhagyható, a **PATH**-ban a nullkomponens a munkajegyzéket jelenti.

A PATH beállításának másik módja, hogy kiterjesztjük az előző értéket:

```
PATH=$PATH:/usr/games          ...Másik lehetőség.
```

* Ez elég szerencsétlenül van a burokban. Jelentősen megnöveli a rendszer terhelését, hogy minden parancs után megnézi a fájlt. Ugyanakkor, ha például sokáig dolgozunk a szövegszerkesztőben, nem értesülünk az új levélről, mert nem adunk a bejelentkezési buroknak parancsokat. Jobb megoldás volna, ha nem minden parancs után, hanem néhány percenként nézné meg. Az 5. és a 7. fejezet leírja, hogyan építhetünk ilyen postaellenőrzőt a rendszerbe. A harmadik lehetőség, amely nem mindenki esetében megvalósítható, hogy maga a **mail** program figyelmeztessen, ha üzenetünk jött; ő ugyanis biztosan értesül róla.

Burokváltozók értékét úgy kaphatjuk meg, hogy neve elé *$* jelet írunk. Fenti példánkban a *$PATH* kifejezés a keresési út aktuális értékét adja, ehhez hozzátesszük az új részt, és az eredményt a **PATH**-ba visszük. Az **echo**-val meggyőződhetünk erről:

```
$ echo PATH is $PATH
PATH is :/bin:/usr/bin:/usr/games
$ echo $HOME                    Bejelentkezési jegyzékünk.
/usr/you
$
```

Ha vannak saját parancsaink, összegyűjthetjük azokat egy saját jegyzékbe, és keresési utunkba felvehetjük ezt a jegyzéket. Példánkban ekkor a **PATH** valami ilyesmi lenne:

```
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
```

Arról, hogy saját parancsokat hogyan készíthetünk, a 3. fejezetben lesz szó.

Egy másik változó, amit az **ed**-nél bonyolultabb szövegszerkesztők gyakran használnak, a **TERM**, ami az alkalmazott terminál típusának nevét tartalmazza. Ez az információ lehetővé teszi a programoknak, hogy hatékonyabban használják a képernyőt. Ezt is beírhatjuk a **.profile** fájlba, valahogy így:

```
TERM = adm3
```

A változókat a rövidítésekre is felhasználhatjuk. Ha gyakran kell egy hosszú nevű jegyzékre hivatkoznunk, **.profile** fájlunkat érdemes egy ilyen sorral bővíteni:

```
d = /borzasztó/hosszú/jegyzék/név
```

és ezután egyszerűen ezt mondhatjuk:

```
$ cd $d
```

Az általunk létrehozott változókat kisbetűvel szoktuk írni, hogy megkülönböztessük a burok által használt nagybetűkkel jelölt változóktól.

Végül megmondhatjuk a buroknak, hogy a változókat más programokban is használni kívánjuk, ezt az **export** paranccsal tehetjük meg (erre a 3. fejezetben visszatérünk):

```
export MAIL PATH TERM
```

Összefoglalóként itt egy tipikus **.profile** fájl:

```
$ cat.profile
stty erase '^h' -tabs
MAIL=/usr/spool/mail/you
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
```

```
TERM=adm3
b=$HOME/book
export MAIL PATH TERM b
date
who | wc -l
$
```

Távolról sem merítettük ki a burok által nyújtott szolgáltatásokat. Az egyik legfontosabb, hogy új, saját parancsokat építhetünk azáltal, hogy meglevő parancsokat berakunk egy, a burok által feldolgozandó fájlba. Csodálatos, hogy mennyi mindent lehet ezzel az egyszerű mechanizmussal megvalósítani. Ennek részletes elemzését a 3. fejezetben fogjuk elkezdeni.

1.5. Ami a UNIX rendszerből még hátravan

A UNIX rendszerben jóval több van annál, amit eddig ismertettünk, de minden nem férhet el egy könyvben. Az eddigiekkel csupán annyi volt a célunk, hogy megbarátkozzunk a rendszerrel és a kézikönyvvel. Ha kérdéseink vannak egy-egy parancs használatával kapcsolatban, a kézikönyvet kell megnéznünk.

A kézikönyvet érdemes csemegézni; felfrissíthetjük, amit a már ismert parancsokról tudunk, és újakat fedezhetünk fel. A kézikönyv egy sor olyan programot ismertet, amit mi nem tárgyalunk. Fordítóprogramokat, mint például a FORTRAN 77, számológép-programokat, mint a **bc(1),** gép – gép közötti kapcsolat létrehozására szolgáló programokat, mint a **cu(1)** és **uucp(1),** grafikus programokat, statisztikai programokat, és mély titkokat, mint amilyen a **units(1).**

Mint már sokszor hangsúlyoztuk, ez a könyv nem helyettesítheti a kézikönyvet, csak kiegészíti azt. A következő fejezetekben a UNIX rendszer elemeit, programjait nézzük át, kiindulva a kézikönyvből, s az egyes részeket összefűző gondolatmenetet követve. Bár a programok közötti kapcsolatok a kézikönyvben sehol nincsenek kifejtve, ezek alkotják a UNIX programozási környezet szövetét.

Történeti és irodalmi megjegyzések

Az eredeti UNIX tanulmányt D. M. Ritchie és K. L. Thompson írta, amely „The UNIX Sharing System" címmel a CACM-ban *(Communications of the ACM)* 1974. júliusában jelent meg. 1983. januárjában újranyomták, ismét CACM-ban. Ezt az áttekintést, amely operációs rendszerek iránt érdeklődőknek készült, minden programozónak érdemes elolvasnia.

A *Bell System Technical Journal* (BSTJ) különszáma (1978. július) számos tanulmányt tartalmaz a UNIX fejlesztésekről, valamint közli Ritchie és Thompson eredeti CACM tanulmányának javított változatát. A BST második különkiadása 1984-ben jelent meg, és újabb UNIX-szal kapcsolatos tanulmányokat tartalmaz.

B. W. Kernighan és S. R. Mashey „The UNIX Programming Environment"

című cikke programozók számára ismerteti a rendszer lényeges jellemzőit (*IEEE Computer Magazin:* 1981. április).

A UNIX Programmer's Manual rendszer változatonként újabb és újabb kiadásban felsorolja a parancsokat, rendszerrutinokat, interfészeket, a fájlformátumokat, karbantartási eljárásokat. Sokáig nem lehetünk meg nélküle, bár a programozás kezdetén elég az első kötet néhány részét olvasni. A 7. kiadás kézikönyvének első kötete Holt, Rinehart és Winston munkája.

A *UNIX Programmer's Manual* második kötetének címe „Documents for use with UNIX Time-sharing System", tartalma a fontosabb parancsok tankönyvei és referenciakönyvei. Ismerteti a dokumentumkészítő programokat és a programfejlesztő eszközöket. Egyszer majd ezeket is el kell olvasnunk!

Ann és Nico Lomuto *A UNIX Primer* című könyve (Prentice-Hall, 1983) jó ismertetés teljesen kezdőknek, különösen azoknak, akik eddig nem programoztak.

2. A fájlrendszer

A UNIX rendszerben minden fájl. Ne higgyük, hogy ez túlzott leegyszerűsítés! Amikor a rendszer első változatát tervezték, amikor még neve sem volt, a figyelmet arra összpontosították, hogy a fájlrendszer világos és könnyen használható szerkezetű legyen. A UNIX rendszer sikerének és kényelmének titka döntően a fájlrendszer. Nagyszerű példája az egyszerűség elvének és annak a hatalmas erőnek, amely néhány jól kiválasztott alapgondolat következetes alkalmazásában rejlik.

Ahhoz, hogy kényelmesen beszélhessünk parancsokról és kapcsolataikról, jól kell ismernünk a fájlrendszer szerkezetének és működésének alapjait. Ebben a fejezetben a fájlrendszer alkalmazásának majdnem minden részletével foglalkozunk: mi a fájl, hogyan adjuk meg; jegyzékek; fájlrendszer-hierarchia; engedélyek; i-bögök (inodes, a rendszernek a fájlokról készített rekordjai)* és eszközfájlok. Mivel a UNIX rendszer használata többnyire fájlok alkalmazásából áll, a rendszerben sok a fájlok vizsgálatára, átrendezésére vonatkozó parancs; ebben a fejezetben a leggyakrabban használtakat mutatjuk be.

2.1. A fájlok alapjai

A fájl bájtok egymásutánja. (A bájt egy kis információmorzsa, általában 8 bit hosszúságú. A mi céljainkra a bájt ugyanaz, mint a karakter.) A rendszer nem kényszerít semmilyen struktúrát a fájlra, és a tartalmának nem tulajdonít semmilyen jelentést, a bájtok jelentése kizárólag a fájlt értelmező programoktól függ. Továbbá mint látni fogjuk, ez nemcsak lemezfájlokra igaz, hanem a perifériákra is.

Mágnesszalagok, postaüzenetek, billentyűzetről bevitt karakterek, nyomtatóra írandó kimenet, adatáramlás csövekben – a rendszer és a benne levő programok számára egyszerűen bájtok egymásutánja.

* Az inode (i-node : information node) kifejezést csakúgy, mint a többi, köznapi nyelvből vett UNIX kifejezést, magyarra fordítottuk. Nyomtatásban használtuk az ékezetes magánhangzókat is, annak ellenére, hogy ma még kevés programban találkozhatunk velük. Azokban a példákban, ahol ezeknek a betűknek a kódszámai előfordultak, a kódszám a szokásos ékezet nélküli helyettesítő betű kódja. *(A szerk.)*

A legjobb tanulási módszer a játék, kezdjük tehát azzal, hogy létrehozunk egy kis fájlt:

```
$ ed
a
itt az idõ
most vagy soha
.
w vacak
26
q
$ ls -l vacak
-rw-r--r--  1 you   26 Sep 27 06:11 vacak
$
```

A **vacak** egy 26 bájtból álló fájl, hiszen 26 karaktert ütöttünk be (kivéve, ha hibáztunk és javítanunk kellett). Nézzük meg, mi van a fájlban:

```
$ cat vacak
itt az idõ
most vagy soha
$
```

A **cat** egyszerűen kiírja a fájl tartalmát, az **od** parancs (octal dump) viszont bájtonként mutatja a fájlt:

```
$ od -c vacak
0000000   i   t   t       a   z       i   d   õ  \n   m   o   s   t
0000020   v   a   g   y       s   o   h   a  \n
0000032
$
```

A **-c** opció azt írja elő, hogy a bájtokat karakterekként értelmezze. A **-b** opciót bekapcsolva a bájtokat oktális formában* (8-as számrendszerben) is kiírja:

```
$ od -cb vacak
0000000   i   t   t       a   z       i   d   õ  \n   m   o   s   t
          151 164 164 040 141 172 040 151 144 157 012 155 157 163 164 040
0000020   v   a   g   y       s   o   h   a  \n
          166 141 147 171 040 163 157 150 141 012
0000032
$
```

* A fájl minden bájtja egy nyomtatható karakter kódja. A legtöbb UNIX rendszerben ASCII kódokat (ASCII : American Standard Code for Information Exchange) alkalmaznak, kivéve néhány, leginkább IBM gyártmányú gépet, amelyeken EBCDIC kódokat (Extended Binary-Coded-Decimal Interchange Code) alkalmaznak. Ebben a könyvben ASCII kódokat tételezünk fel; a karakterek oktális kódját l. **cat /usr/pub/ascii** vagy **ascii (7).**

A bal oldalon látható hétjegyű szám a következő kiírandó karakter helye a fájlban, vagyis sorszáma oktálisan.

Az oktális kiírás szokása a PDP–11 számítógép alkalmazásából ered, az ugyanis az oktális jelölést szereti. Más gépeknél jobb lehet a hexadecimális kiírás, a **–x** opció hatására az **od** hexában dolgozik.

Figyeljük meg, hogy minden sor végén van egy oktális 012 karakter, az ASCII újsor- (newline) karakter; a *RETURN* billentyű megnyomásakor a rendszer ezt teszi a bemenetre. A C nyelvből kölcsönzött jelölési rendszerrel az újsort \n-nel jelöljük. Ezt a jelet az olvasás megkönnyítésére csak az olyan programok használják, mint az **od**. A fájlban tárolt érték egyetlen bájt (012).

Az újsor a leggyakrabban előforduló *különleges karakter*. A terminálvezérléssel kapcsolatos egyéb különleges jelek a visszalépés (oktális 010, kiírt \b), a tab (011, \t) és a kocsivissza (015, \r).

Lényeges különbség van a karakter egy fájlban való tárolása és ugyanannak a karakternek különböző helyzetekben való értelmezése között. Pl. amikor visszalépést ütünk a billentyűzeten, akkor (feltéve, hogy törlőkarakterünk a visszalépés) a mag ezt úgy értelmezi, hogy az előtte beütött karaktert akarjuk törölni. Mind az előző karakter, mind a visszalépés eltűnik, de a visszalépés a terminálra visszhangzódik, és a kurzor eggyel visszalép.

Ha az alábbit gépeljük be:

\←

(vagyis \ és utána visszalépés), azt a mag úgy értelmezi, hogy egy visszalépést akarunk a bemeneten, tehát a \-t félreteszi, a 010 bájtot pedig kikergeti a fájlunkba. A terminálra visszhangzódó visszalépés a kurzort a \ fölé ugratja.

Ha visszalépést tartalmazó fájlt íratunk ki, akkor a visszalépés értelmezés nélkül kerül a terminálra, ami egy karakterrel visszalépteti a kurzort. Ha az **od** programmal kiíratunk egy ilyen fájlt, akkor a visszalépés 010 értékű bájtként, vagy a **–c** opcióval \b alakban megjelenik.

A tabulátorok magyarázata hasonló: a bemenő karakter a terminálra visszhangzódik és a terminálról olvasó program fogadja, a kimenő karaktert egyszerűen a terminál értelmezi. Van azonban egy különbség: megmondhatjuk a magnak, hogy ő értelmezze számunkra a tabulátorokat, ekkor minden kiírandó tabulátor helyett annyi szóközt vesz a mag, hogy a kiírás eljusson a következő tabulátorstopig. A tabulátorstopokat a 9., 17., 25. stb. oszlopokra állították be. A

```
$ stty –tabs
```

parancs hatására a *terminálunkra kiírandó* tabulátorok helyére szóközök kerülnek. (Lásd stty (1).)

A *RETURN* kezelése hasonló. A mag a *RETURN*-t kocsivissza–újsor-ként visszhangozza, de a bemenetre csak az újsort tárolja. A kimenő újsor-karaktert pedig kiegészíti kocsivissza–újsorra.

A UNIX rendszer eléggé szokatlan módon jeleníti meg a vezérlőinformációkat, különösen szokatlan a sorokat lezáró újsor-karakterek használata. Sok rendszer

ehelyett rekordokat alkalmaz, egy sor egy rekord, amely nemcsak az adatainkat tartalmazza, hanem a sorban levő karakterek számát is (de újsort nem). Más rendszerekben minden sor végén kocsivissza- *és* újsor-karakter van, mert a legtöbb terminálra íráskor ezeket kell a kimenetre tenni. (A soremelés ugyanazt jelenti, mint az újsor; a kocsivissza – soremelés : carriage return – linefeed szokásos rövidítése CRLF.)

A UNIX rendszer egyiket sem teszi – nincsenek rekordok, nincsenek rekordszámok, a fájlokban nincs egyetlen olyan bájt sem, amit nem mi vagy a programjaink tesznek oda. Az újsor-karakterből kocsivissza és újsor lesz, ha a terminálra küldjük, a programoknak azonban csak az újsor-karakterrel kell foglalkozniuk, ők csak ezt az egyetlen karaktert látják. A legtöbbször éppen erre az egyszerű szerkezetre van szükség. Ha egy bonyolultabb szerkezetre van szükségünk, azt erre egyszerűen ráépíthetjük; fordítva viszont, a bonyolultból az egyszerűt sokkal nehezebb előállítani.

Minthogy a sor végét az újsor-karakter jelzi, azt hihetnők, hogy a fájl vége egy másik speciális karakter, mondjuk \e. Ha megnézzük az **od** listáját, nem látunk különleges karaktert a fájl végén – egyszerűen csak vége van. A rendszer nem különleges jelből, hanem egyszerűen abból ismeri fel a fájl végét, hogy nincs több adat. A mag tárolja a fájlhosszúságokat, így a programok akkor érzékelik a fájl végét, ha feldolgozták a fájl összes bájtját.

A programok a fájlokban levő adatokat a mag egy szubrutinjával, a **read** nevű rendszerhívással nyerik vissza. A **read** minden meghívásakor visszaadja a fájl következő részét – pl. a terminálon beírt szöveg következő sorát. A **read** azt is megmondja, hogy hány bájtot adott vissza, a fájl végét abból vehetjük észre, hogy a **read** nulla bájtot ad vissza. Ha van még bájt a fájlban, akkor a **read** visszaad valamennyit. Nincs értelme annak, hogy a fájl végét egy speciális bájt értékkel jelezzük, mert mint korábban mondtuk, a bájtok jelentése a fájl értelmezésétől függ. Mivel *minden* fájlnak van vége, és mivel minden fájlt csak a **read**-en keresztül lehet elérni, a nulla visszaadása speciális karakter bevezetése nélkül, értelmezéstől függetlenül jelzi a fájl végét.

Amikor egy program a terminálról olvas, a mag a bemenő sort csak akkor adja át a programnak, ha újsort írunk (vagyis megnyomjuk a *RETURN*-t). Ezért, ha gépelési hibát vétünk, és észrevesszük, mielőtt újsort adunk, visszamehetünk és kijavíthatjuk. Ha már beírtuk az újsort, akkor a rendszer már beolvasta a sort és már nem tudjuk javítani.

A soronkénti bemenet működését jól mutatja a **cat,** amely alapértelmezésben a hatékonyság érdekében pufferezi a kimenetet, de a cat –u pufferezés nélkül, amint beolvasta, ki is írja:

```
$ cat          cat pufferezett kimenettel.
123
456
789
ctl-d
123
456
789
$ cat -u       cat nem pufferezett kimenettel.
123
123
```

```
456
456
789
789
ctl-d
$
```

A **cat** minden sort akkor kap meg, amikor megnyomjuk a *RETURN*-t, pufferezés nélkül, amint megkapja az adatot, kiírja.

Most próbáljunk valami mást: írjunk be néhány karaktert, majd *RETURN* helyett *ctl-d*-t:

```
$ cat -u
123ctl-d123
```

a **cat** azonnal kiírja a karaktereket. A *ctl-d* azt jelenti, hogy a begépelt karaktereket azonnal küldje a terminálunkról olvasó programnak. Maga a *ctl-d* nem kerül a programhoz, ellentétben az újsor-karakterrel, amelyet megkap a program. Most írjunk be még egy *ctl-d* karaktert és semmi mást:

```
$ cat -u
123ctl-d123ctl-d$
```

A burok a $ jellel válaszol, mert a **cat** egyetlen karaktert sem olvasott és ezt úgy értelmezi, hogy vége a fájlnak és megáll. A *ctl-d* elküldi a terminálról beolvasó programnak mindazt, amit beírtunk. Ha nem írtunk be semmit, akkor a program nem olvas be karaktereket, és ez olyan, mintha fájl vége lenne. Ezért van, hogy a *ctl-d* kijelentkezést okoz – a burok úgy látja, hogy nincs több bemenet. A *ctl-d*-t persze általában fájl vége jelnek használjuk, de érdekes, hogy sokkal általánosabb funkciója is van.

2-1. feladat. Mi történik, ha az **ed**-nek *ctl-d*-t adunk?
Hasonlítsuk ezt össze a

```
$ ed <file
```

paranccsal! □

2.2. Mi van a fájlban?

A fájl formáját a fájlt használó program határozza meg. Sokféle program van, ezért a fájltípusok száma is nagy. A fájltípusok azonban a fájlrendszerben nem rögzítettek, így a mag nem tudja megmondani a fájl típusát: nem tudja azt. A **file** parancs megpróbálja kitalálni (röviden elmagyarázzuk, hogyan):

```
$ file /bin/bin/ed /usr/src/cmd/ed.c /usr/man/man 1/ed.1
/bin:                   directory
/bin/ed:                pure executable
/usr/src/cmd/ed.c:      c program text
/usr/man/man 1/ed. 1:   roff, nroff, or egn input text
$
```

E négy különböző típusú fájl mindegyikének van valami köze a szövegszerkesztőhöz: az azt tartalmazó jegyzék (/bin), maga a „bináris" vagy futtatható program (/bin/ed), a programot megadó „forrás" vagy C utasítások (/usr/src/cmd/ed.c) és a kézikönyv megfelelő lapja (/usr/man/man1/ed.1).

A **file** a fájltípus meghatározásakor nem nézi a fájlneveket (bár megtehetné), mert a névkonvenciók csupán konvenciók, ezért nem teljesen megbízhatóak. Pl. a **.c**-re végződő nevű fájlok rendszerint C forráskódok, de semmi sem tiltja, hogy **.c** fájlunkba valami másféle dolgot tegyünk. A **file** nem a nevet nézi, hanem beolvassa a fájl első néhány száz bájtját, és megpróbálja megfejteni a fájl típuskulcsát. (A későbbiekben megmutatjuk, hogy a speciális rendszertulajdonságokkal felruházott fájlok – ilyenek a jegyzékek – a rendszert megkérdezve azonosíthatók, a **file** azonban úgy tud egy jegyzéket azonosítani, hogy beleolvas.)

A fájltípus kulcsa gyakran egyszerűen megtalálható. A futtatható program elejét egy „bűvös" bináris szám jelöli. Az opciók nélküli **od** a fájlt 16 bites (2-bájtos) szavakban írja ki, és láthatóvá teszi a „bűvös" számot:

```
$ od /bin/ed
0000000 000410 025000 000462 011444 000000 000000 000000 000001
0000020 170011 016600 000002 005060 177776 010600 162706 000004
0000040 016616 000004 005720 010066 000002 005720 001376 020076
...
$
```

Az oktális 410 érték tiszta végrehajtható programot jelöl, amelynek végrehajtási kódja több folyamatra bontódhat. (A bűvös számok rendszerfüggőek.) A 410 bitminta nem ASCII szöveg, tehát véletlenül sem állhatott elő egy szövegszerkesztő jellegű programból. Saját programunk futtatásával természetesen létrehozhatunk ilyen fájlt, a rendszer azt bináris programfájlnak fogja érteni.

Szövegfájloknál a kulcs mélyebben lehet elrejtve, ilyenkor a **file** olyan szavak után kutat, mint **#include,** amivel a C nyelvű forrást vagy ponttal kezdődő sorokat keres, amivel az **nroff** vagy a **troff** bemenetet tudja azonosítani.

Vajon miért nem követi a rendszer gondosabban a fájltípusokat, pl. miért nem tiltja, hogy **/bin/ed** a **sort** parancs bemenete legyen? Ennek egyik oka az, hogy el akarja kerülni a hasznos feldolgozási lehetőségek esetleges kizárását. Bár a

```
$ sort /bin/ed
```

parancsnak nincs sok értelme, de sok olyan parancs van, amely bármilyen fájlra működhet. Nincs okunk arra, hogy lehetőségeiket korlátozzuk. Az **od**, **wc**, **cp**, **cmp**, **file**, és sok más folyamat a fájlokat tartalmuktól függetlenül dolgozza fel. A formázat-

lanság gondolata azonban ennél sokkal mélyebbre megy. Ha mondjuk az **nroff** bemenetét megkülönböztetnénk egy C nyelvű forrástól, akkor a szövegszerkesztőt arra kellene kényszerítenünk, hogy létrehozáskor – és valószínűleg módosításkor is – különböztesse meg ezeket. Ha így volna, sokkal nehezebben tudtuk volna a 6...8. fejezetek C nyelvű programjainak listáit elkészíteni.

A UNIX rendszer nem gyártja a különbözőségeket, hanem igyekszik eltüntetni azokat. Minden szöveg újsor-karakterrel lezárt sorokból áll, a legtöbb program érti ezt az egyszerű formát. Könyvünk írásakor sokszor futtattunk szövegfájlokat létrehozó parancsokat, a szövegfájlokat sokszor dolgoztuk fel a fentiekhez hasonló parancsokkal és a szövegszerkesztővel beválogattuk azokat a könyvet kiíró **troff** bemenetére. A majd minden oldalon található másolatokat ilyen parancsokkal állítottuk elő:

```
$ od -c vacak >múlandó
$ ed sz2.1
1534
r múlandó
168
...
```

Az **od** szabványos kimenetére küldi a szöveget, amit azután bárhol felhasználhatunk, ahol szöveget lehet használni. Ez az egységesség szokatlan; a legtöbb rendszer – még szövegekre is – sokféle fájlformát használ, és a programnak vagy a felhasználónak előzetesen kell döntenie arról, hogy a rendszer milyen típusú fájlt hozzon létre. A UNIX rendszerben csak egyféle fájl van, és a fájl eléréséhez csupán annak nevére van szükség.*

A fájlformák sokféleségének hiánya általában előnyös – a programozóknak nem kell foglalkozniuk a fájl típusával, a szabványos programok minden fájlra működnek – de vannak hátrányai is. A rendező-, kereső- és szerkesztőprogramok valóban szöveget várnak bemenetként: bináris fájlokat a **grep** nem tud pontosan vizsgálni, a **sort** nem tud rendezni, a szövegszerkesztők nem tudják kezelni.

A bemenetként szöveget váró programok legtöbbjének vannak korlátai. Sok programot kipróbáltunk egy 30 000 bájtos szövegben, amiben nem volt újsor-karakter. Meglepően kevés viselkedett rendesen, mert a programok nagy része kimondatlanul feltételez egy maximális sorhosszúságot (kivételként l. a sort(1) BUGS szakaszát).

A nem szöveges fájloknak is megvan a saját helyük. Pl. a gyors hozzáférés végett adatbázisok különleges címinformációt igényelnek, ennek a hatékonyság miatt bináris információnak kell lennie. A nem szöveges fájlokhoz azonban a kiszolgáló programok egész családját kell elkészítenünk ahhoz, hogy ugyanazokat a dolgokat meg tudjuk csinálni, amelyeket a szöveges fájlokra az alapprogramok tudnak. A szöveges fájlok a processzoridőt nézve kevésbé hatékonyak, a speciális formátumokat kezelő extra szoftver viszont költséges. Ha fájlformát tervezünk, gondolkodjunk, mielőtt nem szöveges megjelenítést választunk. (Gondolkozzunk akkor is, ha programunk a hosszú bemenő sorok miatt túlságosan nagy lett!)

* Van egy Doug McIlroytól származó jó teszt a fájlrendszer egységességének vizsgálatára. Ezt a tesztet a UNIX könnyedén veszi. Lehet egy FORTRAN program kimenete egy FORTRAN fordító bemenete? Sok rendszernek gondot okozna ez!

2.3. Jegyzékek és fájlnevek

Minden fájlunknak egyértelmű neve van, aminek az eleje /usr/you. Ha csak a **vacak** nevű fájlunk van és **ls** parancsot adunk, akkor azt nem **/usr/you/vacak** alakban írjuk ki, hanem az eleje nélkül csak a fájlnév jelenik meg:

```
$ ls
vacak
$
```

Ez azért van, mert minden futó programnak, vagyis minden folyamatnak van egy *munkajegyzéke*, és minden fájlnévről alapértelmezés szerint azt feltételezi a rendszer, hogy ennek a jegyzéknek a nevével kezdődik, hacsak nincs előtte fordított törtvonal. A bejelentkezési buroknak és az **ls** programnak van tehát egy munkajegyzéke. A **pwd** parancs („print working directory") megnevezi a munkajegyzéket:

```
$ pwd
/usr/you
$
```

A munkajegyzék a folyamatokhoz kapcsolódik, nem a személyekhez vagy a programokhoz – az embereknek bejelentkezési jegyzékük, a folyamatoknak munkajegyzékük van. Ha egy folyamat létrehoz egy gyermekfolyamatot, az örökli szülőjének munkajegyzékét. Ha a gyermek új jegyzékre vált, azzal nem hat szülőjére – az ő munkajegyzéke változatlan marad, függetlenül attól, hogy mit rakoncátlankodik a gyermeke.

A munkajegyzék kényelmes jelölési módszer, hiszen sok gépelést lehet vele megtakarítani, igazi jelentősége azonban az, hogy jó szervezést tesz lehetővé. Az összetartozó fájlok azonos jegyzékhez tartoznak. A **/usr** rendszerint a felhasználói fájlrendszernek a legfelső jegyzéke. (Az usr a user rövidítése.) A **/usr/you** a bejelentkezési jegyzékünk, amely bejelentkezéskor egyben a munkajegyzék is. Az **/usr/src** a rendszerprogramok forrását, a **/usr/src/cmd** a UNIX programok forrását, a **/usr/src/cmd/sh** a burok (shell) forrásprogramjait tartalmazza és így tovább. Ha új munkát kezdünk, vagy ha összetartozó fájljaink vannak, mondjuk néhány recept, akkor az **mkdir** paranccsal létrehozhatunk egy új jegyzéket és abba írhatjuk a fájlokat:

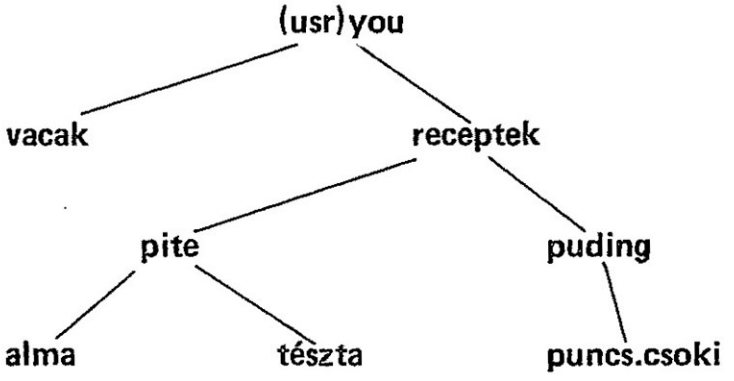
```
$ pwd
/usr/you
$ mkdir receptek
$ cd receptek
$ pwd
/usr/you/receptek
$ mkdir pite puding
$ ed pite/alma
...
$ ed puding/puncs.csoki
...
$
```

Vegyük észre, milyen egyszerűen lehet az aljegyzékekre hivatkozni. A **pite/alma** jelentése nyilvánvaló: ez minden bizonnyal egy almáspite receptje lehet, mégpedig a **/usr/you/pite/alma** jegyzékben. Tehettük volna ezt a receptet a **receptek/alma.pite** fájlba, nem pedig a **recept** aljegyzékbe, de jobb szervezésnek tűnik, ha egy helyre tesszük a pitéket. Így pl. megtehetjük, hogy a tészta receptjét nem írjuk be minden tortareceptbe, hanem külön tároljuk a **receptek/pite/tészta** fájlban.

Bár a fájlrendszer hatékony szervezési eszköz, mégis elfelejthetjük, hogy hova tettük a fájlt, vagy hogy milyen fájljaink vannak. Kézenfekvő, hogy legyen egy-két, a jegyzékekben böngészgető parancs. Az **ls** parancs segít a fájlokat keresni, de az aljegyzékbe nem kukkant bele.

```
$ cd
$ ls
vacak
receptek
$ file *
vacak:   ascii text
receptek:   directory
$ ls receptek
pite
puding
$ ls receptek/pite
alma
tészta
$
```

A fájlrendszernek ezt a részét így ábrázolhatjuk:



A **du** (disc usage : lemezhasználat) parancs megmondja, hogy mennyi helyet foglalnak el a jegyzékben levő fájlok, beleértve az aljegyzékekben levőket is.

```
$ du
 6      ./receptek/pite
 4      ./receptek/puding
11      ./receptek
13      .
$
```

A fájlnevek világosak, a számok pedig a fájlok tárolásához szükséges lemezblokkok számát adják; rendszerint 512 vagy 1024 bájt egy blokk. A jegyzékhez tartozó számérték azt mutatja, hogy a jegyzékben és annak aljegyzékeiben levő fájlok együtt mennyi helyet foglalnak el, beleértve magát a jegyzéket is.

A **du** parancs a **−a** opcióval (all : összes) a jegyzékben levő összes fájlt kiírja. Ha közülük valamelyik történetesen egy jegyzék, akkor azt is feldolgozza:

```
$ du −a
 2        ./receptek/pite/alma
 3        ./receptek/pite/tészta
 6        ./receptek/pite
 3        ./receptek/puding/puncs.csoki
 4        ./receptek/puding
11        ./receptek
 1        ./vacak
13        .
$
```

Ha a **du −a** kimenetét egy csövön átadjuk a **grep** parancsnak, akkor megkereshetünk egy adott fájlt:

```
$ du −a | grep puncs
3         ./receptek/puding/puncs.csoki
$
```

Az 1. fejezetben láttuk, hogy a **.** név a munkajegyzéket jelenti. Ez lehetővé teszi, hogy egy jegyzékhez hozzáférjünk anélkül, hogy ismernénk a teljes nevét. A **du** fájlokat keres egy jegyzékben. Ha nem mondjuk meg, melyik jegyzékben keressen, akkor a **.** nevet, azaz a munkajegyzék nevét tételezi fel. Ezért a **vacak** és a **./vacak** ugyanannak a fájlnak a nevei.

A jegyzékek a magon belüli alapvető szerepüktől eltekintve ugyanolyanok a fájlrendszerben, mint a közönséges fájlok. Ugyanúgy olvashatók, mint azok, de nem lehet közönséges fájlok módjára létrehozni őket vagy írni rájuk.

Eljött az idő, hogy bepillantsunk egy jegyzék bájtjaiba:

```
$ od −cb.
0000000    4   ;   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
         064 073 056 000 000 000 000 000 000 000 000 000 000 000 000 000
0000020  273   (   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
         273 050 056 056 000 000 000 000 000 000 000 000 000 000 000 000
0000040  252   ;   r   e   c   e   p   t   e   k  \0  \0  \0  \0  \0  \0
         252 073 162 145 143 145 160 164 145 153 000 000 000 000 000 000
0000060  230   =   v   a   c   a   k  \0  \0  \0  \0  \0  \0  \0  \0  \0
         230 075 166 141 149 141 153 000 000 000 000 000 000 000 000 000
0000100
$
```

Látjuk, hol rejtőznek a fájlnevek? A jegyzék bináris és szöveges adatok kombinációja, 16 bájtos darabokból áll, amelyek utolsó 14 bájtja tartalmazza a fájlnevet, ASCII nullákkal (0 értékekkel) feltöltve, az első két bájt pedig megmondja a rendszernek, hol találhatók a fájlra vonatkozó adminisztratív információk – később erre még visszatérünk. Minden jegyzék két elemmel : a . és a .. -tal kezdődik.

```
$ cd                                    Haza.
$ cd receptek
$ pwd
/usr/you/receptek
$ cd .., pwd                            Egy szinttel feljebb.
/usr/you
$ cd ..; pwd                            Még egy szinttel feljebb.
/usr
$ cd ..; pwd                            Még egy szinttel feljebb.
/
$ cd ..; pwd                            Még egy szinttel feljebb.
/                                       Nem lehet feljebb menni.
$
```

A / jegyzék a fájlrendszer *gyökere* (root). Minden fájl a gyökérjegyzékhez vagy annak valamelyik aljegyzékéhez tartozik. A gyökér saját magának a szülőjegyzéke.

2-2. feladat. Az ebben a szakaszban elmondott információk alapján nagyjából értenünk kell, hogyan működik az **ls** parancs. Útmutatás: *cat . >foo; ls −f foo.* □

2-3. feladat. (Nehéz!) Hogyan működik a **pwd** parancs? □

2-4. feladat. A **du** parancs arra készült, hogy a lemezhasználatot figyelje. Ezzel a paranccsal fájlt keresni a jegyzékhierarchiában elég fura ötlet, szinte lehetetlen. Jobb megoldás erre a **find(1)** parancs; nézzük meg a kézikönyvben és hasonlítsuk össze a kettőt! Különösen a **du −a | grep ...** parancsokat hasonlítsuk össze a megfelelő **find** hívással! Melyik gyorsabb? Mi jobb, egy új eszközt létrehozni vagy egy meglevő határesetét használni? □

2.4. Engedélyek

Minden fájlhoz tartozik egy engedélykészlet, ami meghatározza, hogy ki és mit csinálhat a fájllal. Ha pl. valaki a szerelmes leveleit a rendszerben tartja, mondjuk hierarchikusan elrendezve egy jegyzékben, valószínűleg nem szeretné, ha azokat más is elolvashatná. A pletykálkodást megakadályozhatjuk, ha minden levél engedélyét megváltoztatjuk (vagy csak néhány levélét, hadd gyanakodjanak), vagy megváltoztatjuk a leveleket tartalmazó jegyzék engedélyeit és így távoltarthatjuk a kukucskálókat.

De legyünk óvatosak: minden UNIX rendszerben van egy különleges felhasználó, a szuperfelhasználó (super-user), aki a rendszer *minden* fájlját olvashatja, módosíthatja. A szuperfelhasználó privilégiumai egy speciális bejelentkezési névhez, a **root** névhez kötődnek; ezt használja a rendszeradminisztrátor a rendszer karbantartásakor. Van egy parancs is, a **su**, amellyel szuperfelhasználói státusba kerülhetünk, ha ismerjük a **root** jelszavát. Tehát aki ismeri a szuperfelhasználó jelszót, elolvashatja szerelmes leveleinket, ezért „kényes" anyagot ne tároljunk a rendszerben!

Ha több titkosságot akarunk, a **crypt** paranccsal megváltoztathatjuk a fájlban levő adatokat úgy, hogy még a szuperfelhasználó se tudja elolvasni vagy legalábbis megérteni (l. crypt(1)). Természetesen a **crypt** sem nyújt tökéletes biztonságot. A szuperfelhasználó megváltoztathatja magát a **crypt** parancsot, és agyafúrt módon aláásshatja a cryptalgoritmust. Az előbbi hűtlen kezelés, az utóbbi megvalósítása igen nehéz, úgyhogy a **crypt** gyakorlatilag elég biztonságos.

A valóságban a legtöbbször ott esik csorba a titkosságon, ahol nincsenek vagy könnyen kitalálhatók a jelszavak. Néha a rendszeradminisztrátor hanyagsága teszi lehetővé, hogy egy rosszindulatú felhasználó megszerezhesse a szuperfelhasználói engedélyt. A biztonsággal kapcsolatos kérdéseket részletesebben a fejezet végén levő irodalomjegyzék tanulmányai tárgyalják.

Amikor bejelentkezünk, beírunk egy nevet, majd ellenőrzésként egy jelszót. A név a bejelentkezési azonosítónk (**login-id**). A rendszer valójában egy azonosítószám alapján ismer fel minket, ezt *felhasználói azonosítónak* (*user-id* vagy *uid*) nevezzük. Különböző bejelentkezési azonosítókhoz tartozhat azonos felhasználói azonosító is, ekkor ezek a rendszer számára megkülönböztethetetlenek. Ez azonban elég ritka, és a titkosság szempontjából sem kívánatos. Az **uid** mellett van egy *csoportazonosító* is (**group-id**), amivel a felhasználók különböző osztályokba sorolhatók. Sok rendszerben minden közönséges felhasználó (akiknek nem olyan bejelentkezési azonosítójuk van, mint a **root**) egy *other* nevű osztályba tartozik, de lehet, hogy Olvasóink rendszerén másképp van. A fájlrendszer és így általában a UNIX rendszer megszabja, hogy az **uid** és a csoportazonosító számára biztosított engedélyek mire jogosítanak.

Az **/etc/passwd** fájl a *jelszófájl*, ez tartalmazza a felhasználók összes bejelentkezési információját. A **/etc/passwd** fájlba bepillantva ugyanúgy, ahogy a rendszer is teszi, megkereshetjük **uid**-ünket és csoportazonosítónkat:

```
$ grep you /etc/passwd
you:gkmbCTrJ04COM:604:1:Y.O.A.People:/usr/you:
$
```

A jelszó fájl mezőit kettőspontok választják el, jelentésük az alábbi (l. passwd(5)):

bejelentkezési azonosító: titkosított jelszó: felhasználási azonosító: csoportazonosító: vegyes információ: bejelentkezési jegyzék: burok

A jelszófájl közönséges szöveg, a fájlban levő információt felhasználó programok ismerik a mezők definícióit és az elválasztó karaktert.

A burokmező gyakran üres, ami azt jelenti, hogy az alapértelmezés szerinti

/bin/sh burkot használjuk. A „vegyes információ" mező bármit tartalmazhat, gyakran a nevet, címet vagy telefonszámot írják ide.

Megjegyezzük, hogy a második mezőben a jelszó is rejtjelezett formában jelenik meg. A jelszófájlt mindenki olvashatja (épp az imént olvastunk belőle), ezért ha maga a jelszó lenne itt, bárki használhatná a nevünket. Amikor a **login** programnak megadjuk a jelszót, titkosítja és összehasonlítja az eredményt a **/etc/passwd** fájlban levő titkosított jelszóval. Ha egyezik, megengedi, hogy bejelentkezzünk. Ez azért működik jól, mert a titkosító algoritmus olyan, hogy az eredeti formából a titkosítottat egyszerűen elő lehet állítani, de fordítva nagyon nehéz. Pl. ha a jelszónk *ka-boom*, az rejtjelezve mondjuk *gkmbCTrJ04COM*, de ebből nem egykönnyen lehet az eredetit visszakapni.

A mag úgy dönti el, hogy olvashatjuk-e a /etc/passwd fájlt, hogy megnézi a fájlhoz tartozó engedélyeket. Minden fájlra háromféle engedély adható: olvasás (read; tartalom vizsgálata), írás (write; tartalom megváltoztatása) és végrehajtás (execute; programként futtatás). Különböző személyeknek különböző engedélyeket adhatunk. Az első engedélykészlet (olvasás, írás, végrehajtás) a fájl tulajdonosáé. A csoportnak egy másik engedélykészlete van. Végül a harmadik engedélykészlet a többiekre vonatkozik.

Az **ls** parancs **-l** opciója többek között az engedélyekre vonatkozó információt is kiírja:

```
$ ls -l /etc/passwd
-rw-r--r--  1 root      5115 Aug 30 10 : 40 /etc/passwd
$ ls -lg /etc/ passwd
-rw-r--r--  1 adm       5115 Aug 30 10 : 40 /etc/passwd
$
```

A két sor együtt a következőket mutatja: az **/etc/passwd** fájl bejelentkezési azonosítója **root**, amely az **adm** csoportba tartozik, hossza 5115 bájt, utoljára augusztus 30-án 10.40-kor módosították, és egy láncszemből (link) áll (vagyis egy neve van, ezzel majd a következő szakaszban foglalkozunk). Az **ls** egyes változatai egy hívással kiírják a tulajdonost és a csoportot.

Az **ls** parancs -rw-r--r-- alakban jeleníti meg a fájlra vonatkozó engedélyeket. Az első kötőjel azt jelzi, hogy ez közönséges fájl. Ha jegyzék volna, ezen a helyen d betű állna. A következő három karakter az (**uid** szerinti) tulajdonos olvasási, írási, végrehajtási engedélyeit tartalmazza. A rw- azt jelenti, hogy a **root** (a tulajdonos) olvashatja, írhatja a fájlt, de nem hajthatja végre. Végrehajtható fájl esetén itt x állna.

A következő három karakter (r--) a csoportengedélyeket tartalmazza, ami ebben az esetben az **adm** csoport tagjaira – feltehetően a rendszeradminisztrátorokra – vonatkozik, ők olvashatják a fájlt, de nem írhatják és nem futtathatják. A következő három karakter (ez is r--) a többiek – a rendszer összes többi felhasználója – engedélyei. Ezen a gépen tehát csak a **root** változtathatja meg a felhasználók bejelentkezési információját, de bárki elolvashatja a fájlt, megismerheti a benne levő információt. Elfogadható lenne az is, ha az **adm** csoport írásengedélyt is kapna az **/etc/passwd** fájlhoz.

A **/etc/group** fájl a csoportneveket és csoportazonosítókat adja meg, és meghatározza, hogy melyik felhasználó melyik csoporthoz tartozik. Az **/etc/passwd** fájl a bejelentkező csoportját azonosítja; csoportengedélyeinket a **newgrp** paranccsal változtathatjuk meg.

Bárki mondhatja, hogy

```
$ ed /etc/passwd
```

és szerkesztheti a jelszófájlt, de változtatásokat csak a **root** írhatja vissza. Hogyan változtathatjuk meg saját jelszónkat annak ellenére, hogy nincs engedélyünk a jelszófájl javítására? Erre szolgál a **passwd** program, amelyet valószínűleg a **/bin** jegyzékben találunk meg:

```
$ ls -l /bin/passwd
-rwsr-xr-x 1 root          8454 Jan 4 1983 /bin/passwd
$
```

(Megjegyezzük, hogy az **/etc/passwd** a bejelentkezési információt tartalmazó szöveges fájl, a **/bin/passwd** egy másik jegyzékben levő végrehajtható fájl, ami lehetővé teszi, hogy a jelszóinformációt megváltoztassuk.) Az engedélyek ebben az esetben azt mutatják, hogy bárki végrehajthatja a parancsot, de csak a **root** változtathatja meg azt. A fájl tulajdonosának végrehajtásiengedély-mezőjében az x helyén álló s („set-uid" : felhasználói azonosító állítása) azt jelenti, hogy amikor a parancs éppen végrehajtás alatt áll, akkor megkapja a fájl tulajdonosának, esetünkben a **root**-nak az engedélyeit. A **/bin/passwd** futtatásakor a felhasználói azonosító **root**-tá válik, így a **passwd** paranccsal bármely felhasználó szerkesztheti a jelszófájlt.

A *set-uid* egyszerű és nagyszerű ötlete* számos titkosítási probléma megoldása. Pl. egy játékprogram írója a program tulajdonosának **s** engedélyt adhat, így a felhasználók fel tudják frissíteni a program eredményfájlját, ami egyébként hozzáférhetetlen volna számukra. A *set-uid* veszélyes is lehet A **/bin/passwd** fájlnak nagyon pontosnak kell lennie; ha nem az, akkor tönkre lehet tenni a **root** védelme alatt álló rendszerinformációkat. Ha annak engedélyei -rwsrwxrwx lennének, akkor *bármelyik* felhasználó felülírhatná, és a fájlt bármilyen más programra kicserélhetné. Ez különösen veszélyes a felhasználói azonosítót beállító programokra, hiszen a **root** a rendszer minden fájljához hozzáférhet. (Némelyik UNIX rendszer fájlmódosításkor kikapcsolja a *set-uid* bitet, hogy csökkentse a biztonsági hiány veszélyét.)

A *set-uid* bit hatásos eszköz, de elsősorban néhány olyan rendszerprogramnál használatos, mint a **passwd**. Nézzünk egy egyszerűbb fájlt!

```
$ ls -l /bin/who
-rwxrwxr-x 1 root          6348 Mar 29 1983 /bin/who
$
```

A **who** parancsot a **root** és a tulajdonos csoportja írhatja. A *végrehajtható* a következőt jelenti: amikor beírjuk a

```
$ who
```

* A *set-uid bit* Dennis Ritchie szabadalma.

parancsot, akkor a burok a **who** nevű fájlt keresve végignéz egy sereg jegyzéket, amelyek közül az egyik a **/bin**. Ha talál ilyen fájlt, és ha annak van végrehajtási engedélye, akkor a burok hívja a magot, hogy futtassa azt. Megjegyezzük, hogy egy program egyszerűen egy olyan fájl, aminek végrehajtási engedélye van. A következő fejezetben bemutatunk olyan programokat, amelyek egyszerű szövegfájlok, de parancsokként futtathatók, mert van végrehajtási engedélyük.

A jegyzékengedélyek egy kicsit másként működnek, de az alapötlet hasonló.

```
$ ls -ld .
drwxrwxr-x 3 you          80 Sep 27 06 : 11 .
$
```

Az **ls** parancs **-d** opciója magáról a jegyzékről tájékoztat és nem a tartalmáról, a kiírt sor elején álló d azt jelzi, hogy a . valóban jegyzék. Az r mező azt jelenti, hogy olvashatjuk a jegyzéket, az **ls** (vagy az **od**) paranccsal megnézhetjük, milyen fájlok vannak benne. A w azt jelenti, hogy ebben a katalógusban létrehozhatunk és törölhetünk fájlokat, ezek a tevékenységek a jegyzékfájl módosítását, ezért írását jelentik.

Valójában nem lehet egyszerűen beleírni egy jegyzékbe, még a **root**-nak is tilos!

```
$ who >.                  Próbáljuk a . -ot átírni!
.: cannot create          Nem tudjuk.
$
```

A fájlok létrehozására és törlésére rendszerhívások vannak, csak ezeken keresztül lehet a jegyzékek tartalmát megváltoztatni. Az engedélyeknek itt is megvan a szerepük, a w mezők mondják meg, ki használhatja a rendszerrutinokat a jegyzék módosítására.

A fájl törlésének engedélye a fájltól független. Ha egy jegyzékben írásengedélyünk van, akkor az abban levő fájlokat még akkor is törölhetjük, ha nincs rájuk írás engedélyünk. Az **rm** parancs megerősítést kér, mielőtt törölné a védett fájlt, ezzel ellenőrzi, valóban ezt akarjuk-e tenni – egyike azon ritka alkalmaknak, amikor egy UNIX program kétszer ellenőrzi szándékunkat. (Az **rm** az **-f** opcióval kérdezősködés nélkül töröl.)

A jegyzék engedélyeinek x mezője nem végrehajtást, hanem keresést jelent. A jegyzékre adott végrehajtási engedély azt határozza meg, hogy ki kereshet fájlt a jegyzékben. Ezért, ha egy jegyzékre a többi felhasználó számára – – x engedélyeket adunk, akkor a felhasználók hozzáférhetnek az általuk ismert fájlokhoz, de nem futtathatják rajta az **ls** programot, hogy megnézzék, milyen fájlok vannak a jegyzékben. Hasonlóan, ha egy jegyzékre r – – engedélyeket adunk, akkor a felhasználók az **ls** paranccsal megnézhetik, mi van benne, de nem használhatják annak tartalmát. Egyes rendszereken a túlterhelt időszakokra ezzel a módszerrel kapcsolják ki a **/usr/games** jegyzéket.

A fájlokra adott engedélyeket a **chmod** (change mode) paranccsal változtathatjuk meg:

```
$ chmod engedélyek fájlnevek...
```

Az *engedélyek* szintaktikája ügyetlen. Kétféleképpen adhatjuk meg: oktálisan vagy szimbolikus jelöléssel. Az oktális számok használata egyszerűbb, bár néha kellemesebb a szimbolikus leírás, mert relatív változtatásokat adnak meg. Jó lenne, ha így is megadhatnánk:

```
$ chmod rw-rw-rw- vacak        Nem működik!
```

nemcsak így:

```
$ chmod 666 vacak
```

de sajnos nem lehet.

Oktálisan az olvasáshoz 4-et, az íráshoz 2-t, a végrehajtáshoz 1-et kell összeadnunk. A három számjegy ugyanúgy, mint az **ls**-nél, sorra a tulajdonos, a csoport és a többiek engedélyeit adja. A szimbolikus jelölést nehéz elmagyarázni, ezért nézzük meg a chmod(1) leírást! A mi céljainkhoz elég annyi, hogy a + bekapcsol, a − kikapcsol egy engedélyt. Pl.:

```
$ chmod +x parancs
```

mindenkinek megengedi, hogy végrehajtsa a *parancs*-ot és a

```
$ chmod -w fájl
```

mindenki írásengedélyét kikapcsolja, még a tulajdonosét is. A szuperfelhasználó kivételével csak a fájl tulajdonosa változtathatja meg a fájl engedélyeit, függetlenül maguktól az engedélyektől. Még az írásengedély sem jogosít arra, hogy az engedélybiteket megváltoztassuk.

```
$ ls -ld /usr/mary
drwxrwxrwx 5 mary        704 Sep 25 10 : 18 /usr/mary
$ chmod 444 /usr/mary
chmod: can't change /usr/mary
$
```

Ha viszont a jegyzékre van írásengedély, akkor a fájlokra adott engedélyektől függetlenül bárki törölheti a fájlokat. Ha biztosítani akarjuk, hogy se magunk, se a barátaink ne törölhessenek fájlt egy jegyzékünkből, akkor töröljük annak írásengedélyét:

```
$ cd
$ date >múlandó
$ chmod -w .                   A jegyzéket írhatatlanná teszi.
$ ls -ld .
dr-xr-xr-x 3 you               80 Sep 27 11 : 48 .
$ rm múlandó
rm: múlandó not removed        Nem lehet törölni a fájlt.
$ chmod 775 .                  Visszaadjuk az engedélyt.
$ ls -ld .
```

```
drwxrwxr-x 3 you          80 Sep 27 11 : 48 .
$ rm múlandó
$                                    Most már megy.
```

és **múlandó** már el is tűnt. Megjegyezzük, hogy a jegyzék engedélyeinek átírása nem változtatja meg annak módosítási dátumát. A módosítási dátum csak akkor változik, ha a fájl tartalmában nem csupán az engedélyekben történik változás. Az engedélyek és dátumok nem magában a fájlban tárolódnak, hanem egy *index-bög* (vagy i-bög, *i-node*) nevű rendszerstruktúrában, amelyet a következő szakaszban tárgyalunk.

2-5. feladat. Kísérletezzünk a **chmod**-dal! Próbálkozzunk különböző egyszerű engedélymódokkal, mint a *0* vagy az *1*. Vigyázzunk, nehogy tönkretegyük a bejelentkezési jegyzékünket! □

2.5. Az i-bögök

A fájloknak több összetevőjük van: név, tartalom és olyan adminisztratív információk, mint az engedélyek és a módosítási idők. Az adminisztratív információk az olyan fontos rendszeradatokkal, mint a fájl hosszúsága, tárolási helye a lemezen stb. együtt az i-bögben tárolódnak (i-node : információcsomó).

Az i-bög három időt tárol: a fájltartalom utolsó változtatásának (írásának) idejét, az utolsó használat (olvasás vagy futtatás) idejét és az i-bög utolsó változtatásának (pl. az engedélyek átállításának) az idejét.

```
$ date
Tue Sep 27 12 : 07 : 24 EDT 1983
$ date >vacak
$ ls -l vacak
-rw-rw-rw- 1 you          29 Sep 27 12 : 07 vacak
$ ls -lu vacak
-rw-rw-rw- 1 you          29 Sep 27 06 : 11 vacak
$ ls -lc vacak
-rw-rw-rw- 1 you          29 Sep 27 12 : 07 vacak
$
```

A fájl tartalmának változtatása nem érinti az **ls -lu** által jelzett utolsó használat idejét, az engedélyek megváltoztatása csak az **ls -lu** által jelzett i-bög változtatási idejét érinti.

```
$ chmod 444 vacak
$ ls -lu vacak
-r--r--r-- 1 you          29 Sep 27 06 : 11 vacak
$ ls -lc vacak
-r--r--r-- 1 you          29 Sep 27 12 : 11 vacak
$ chmod 666 vacak
$
```

A **ls** parancs **−t** opciója alapértelmezésben az utolsó módosítás időpontja szerint rendezi a fájlokat, ugyanez a **−c** vagy **−u** opcióval együtt az i-bög utolsó módosításának, ill. a fájl utolsó olvasásának időpontja szerint listáz:

```
$ ls receptek
pite
puding
$ ls −lut
total 2
drwxrwxrwx   4 you        64 Sep 27 12 : 11 receptek
−rw−rw−rw 1 you        29 Sep 27 07 : 11 vacak
$
```

Látjuk, hogy a **receptek**-et használtuk legutóbb, hisz épp az imént néztük meg a tartalmát.

Fontos, hogy megértsük az i-bögöket! Nemcsak az **ls** opciói miatt, hanem azért is, mert szigorúan véve, az i-bögök maguk a *fájlok*. A jegyzékhierarchia csupán kényelmes neveket ad a fájloknak. A rendszer belső fájlnevei az *i-számok*, amelyek a fájlról szóló információt tartalmazó i-bögök sorszámai. Az **ls −i** decimálisan kiírja az i-számot:

```
$ date >x
$ ls −i
15768 vacak
15274 receptek
15852 x
$
```

A jegyzék fájlnév előtti első két bájtja tárolja az i-számot. Az **od −d** oktális bájtok helyett decimális bájtpárokban írja ki az adatokat, és így az i-szám láthatóvá válik:

```
$ od −c .
0000000   4   ;   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 273   (   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040 252   ;   r   e   c   e   p   t   e   k  \0  \0  \0  \0  \0  \0
0000060 230   =   v   a   c   a   k  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100 354   =   x  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000120
$ od −d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 25955 29807 27493 00000 00000 00000
0000060 15768 24950 24931 00107 00000 00000 00000 00000
0000100 15852 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

A jegyzékek elején levő bájt az egyetlen kapcsolat a fájl neve és tartalma között. A jegyzékben levő fájlnevet *láncszemnek* (link) nevezzük, mivel ez láncolja hozzá a jegyzékhierarchiában levő nevet az i-böghöz, és így az adathoz. Ugyanaz az i-szám több jegyzékben is megjelenhet. Az **rm** parancs valójában nem az i-bögöket törli, hanem a jegyzékelemet vagy láncszemet. Csak akkor törli az i-bögöt és ezzel magát a fájlt, ha a fájl utolsó láncszeme is eltűnik.

Ha a jegyzékelemben levő i-szám nulla, az azt jelenti, hogy ez a láncszem törlődött, de a fájl tartalma nem feltétlenül – lehet még láncszem valahol máshol. Megbizonyosodhatunk róla, hogy az i-szám a fájl törlésével nulla lesz:

```
$ rm x
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 25955 29807 27493 00000 00000 00000
0000060 15768 24950 24931 00107 00000 00000 00000 00000
0000100 00000 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

Most olyan fájlt fogunk létrehozni ebben a jegyzékben, amelyet nemigen fogunk használni, de amelynek valószínűleg más lesz az i-száma.

A **ln** parancs egy már létező fájlhoz láncszemet létesít:

```
$ ln régi_fájlnév új_fájlnév
```

A láncszem arra jó, hogy ugyanannak a fájlnak két nevet adjunk, a nevek gyakran két különböző jegyzékben vannak. Sok rendszeren a **/bin/ed** fájlnak **/bin/e** láncszeme is van, így az editor az **e** paranccsal is behívható. Az egy fájlhoz tartozó két láncszem ugyanarra az i-bögre mutat, így azonos lesz az i-számuk is.

```
$ ln vacak vacaklánc
$ ls -li
total 3
15768 -rw-rw-rw-  2 you     29 Sep 27 12 : 07 vacak
15768 -rw-rw-rw-  2 you     29 Sep 27 12 : 07 vacaklánc
15274  drwxrwxrwx  4 you     64 Sep 27 09 : 34 receptek
$
```

Az engedélyek és a tulajdonos között szereplő egész szám a láncszemek száma. Miután mindegyik láncszem az i-bögre mutat, mindegyik láncszem egyformán fontos – nincs különbség az első és a rákövetkező között. (Megjegyezzük, hogy az **ls** rosszul számolja ki a lemezterületet, mert duplán számlál.)

Ha fájlt módosítunk, bármelyik nevén férünk hozzá, létrejön a módosítás, hiszen a láncszemek mind ugyanarra a fájlra mutatnak.

```
$ echo x > vacak
$ ls -l
total 3
```

```
-rw-rw-rw-  2 you        2 Sep 27 12 : 37 vacak
-rw-rw-rw-  2 you        2 Sep 27 12 : 37 vacaklánc
drwxrwxrwx  4 you       64 Sep 27 09 : 34 receptek
$ rm vacaklánc
$ ls -l
total 2
-rw-rw-rw-  1 you        2 Sep 27 12 : 37 vacak
drwxrwxrwx  4 you       64 Sep 27 09 : 34 receptek
$
```

Miután a **vacaklánc** fájlt töröltük, a láncszemek száma visszamegy egyre. Amint mondtuk, az **rm**-ezés csak egy láncszemet szüntet meg, a fájl pedig mindaddig megmarad, amíg az utolsó láncszem is eltűnik. A gyakorlatban persze a legtöbb fájlnak egy láncszeme van, de újfent meggyőződhetünk egy egyszerű elgondolás rugalmasságáról.

A gyorskezűek vigyázzanak: ha az utolsó láncszem is megszűnik, az adat visszaszerezhetetlen! A törölt fájlok nem papírkosárba, hanem hamvasztóba kerülnek, és a hamuból nem lehet visszanyerni azokat! (Van egy halovány remény a feltámadásra. A UNIX rendszerek nagy részének van olyan háttértárolási eljárása, amivel időnként biztos helyre, pl. mágnesszalagra másolja a megváltoztatott fájlokat, amelyek innen visszanyerhetők. A magunk védelme és nyugalma érdekében tudnunk kell, hogy van-e ilyen a rendszerünkön. Ha nincs, vigyázzunk – egy lemezhiba és kész a katasztrófa!)

A láncszemek nagyon hasznosak, ha két ember osztozni kíván egy fájlon, de előfordulhat, hogy két különálló másolatot szeretnének – különböző fájlokat ugyanazzal a tartalommal. Másolatot szeretnénk, mielőtt valamit átírunk, hogy az eredetit vissza tudjuk állítani, ha nem tetszenek a változtatások. A láncszemek ebben nem segítenek, mert amikor az adatok változnak, mindkét láncszem mutatja a változásokat. Fájlokat a **cp** paranccsal másolhatunk:

```
$ cp vacak másodvacak
$ ls -li
total 3
15850 -rw-rw-rw-  1 you        2 Sep 27 13 : 13 másodvacak
15768 -rw-rw-rw-  1 you        2 Sep 27 12 : 37 vacak
15274 drwxrwxrwx  4 you       64 Sep 27 01 : 34 receptek
$
```

A **vacak** és a **másodvacak** i-száma különböző, mert különböző fájlok, annak ellenére, hogy a tartalmuk most éppen azonos. Okos gondolat a másolat engedélyeit megváltoztatni, hogy nehezebb legyen azt véletlenül törölni:

```
$ chmod -w másodvacak        Töröljük az írásengedélyt!
$ ls -li
total 3
15850 -r--r--r--  1 you        2 Sep 27 13 : 13 másodvacak
15768 -rw-rw-rw-  1 you       29 Sep 27 12 : 37 vacak
15274 drwxrwxrwx  4 you       64 Sep 27 09 : 34 receptek
$ rm másodvacak
rm: másodvacak 444 mode n    Nem! Nagyon fontos!
```

```
$ date >vacak
$ ls -li
total 3
15850 -r--r--r--  1 you    2 Sep 27 13 : 13 másodvacak
15768 -rw-rw-rw-  1 you    2 Sep 27 12 : 37 vacak
15274 drwxrwxrwx  4 you   64 Sep 27 09 : 34 receptek
$ rm másodvacak
rm: másodvacak 444 mode y        Már nem is olyan fontos.
$ ls -li
total 2
15768 -rw-rw-rw-  1 you   29 Sep 27 13 : 16 vacak
15274 drwxrwxrwx  4 you   64 Sep 27 09 : 34 receptek
$
```

A másolat megváltoztatása nem jelenti az eredeti megváltozását, és a másolat törlése nem törli az eredetit. Megjegyezzük, hogy miután a **másodvacak** írásvédett volt, az **rm** megerősítést kért, mielőtt törölte volna a fájlt.

Van még egy fájlkezelő parancsunk, az **mv,** amely a láncszemek átrendezésével mozgatja vagy átnevezi a fájlt. Szintaxisa ugyanaz, mint a **cp** vagy az **ln** szintaxisa:

```
$ mv vacak ugyanaz
$ ls -li
total 2
15274 drwxrwxrwx  4 you   64 Sep 27 09 : 34 receptek
15768 -rw-rw-rw-  1 you   29 Sep 27 13 : 16 ugyanaz
$
```

Az **ugyanaz** nevű fájl azonos a régi **vacak** fájlunkkal, mint az i-szám is mutatja, csak a neve – a 15768 i-böghöz tartozó név – változott.

Az eddigi fájlátrendezgetéseket egy jegyzéken belül csináltuk, de ugyanígy megy jegyzékek között is. Az **ln** parancsot gyakran használjuk azonos nevek különböző jegyzékekben való elhelyezésére, pl. amikor többen dolgoznak egy programon vagy dokumentumon. Az **mv** át tudja vinni a fájlt vagy jegyzéket az egyik jegyzékből egy másikba. Az **mv** és a **cp** parancsoknak azonos szintaxisa van erre:

```
$ mv (vagy cp) fájl1 fájl2 ... jegyzék
```

átviszi (vagy lemásolja) a fájlokat az utolsó argumentumként megadott jegyzékbe. Pl. ha ujjgyakorlatokat akarunk végezni az editorral, ezzel kezdhetjük:

```
$ cp /usr/src/cmd/ed.c .
```

és lesz egy saját forráspéldányunk, amellyel kedvünkre játszhatunk. Ha a burkon dolgozunk, amelyben számos különböző forrásfájl van, ezt mondhatjuk:

```
$ mkdir sh
$ cp /usr/src/smd/sh/* sh
```

és a **cp** a burokban levő összes forrásfájlról készít egy másolatot az **sh** aljegyzékünkbe.

Néhány rendszeren az **ln** is elfogad több fájlargumentumot, az utolsó argumentum itt is a jegyzéknév. Néhány rendszeren maguk az **mv, cp, ln** parancsok is egyetlen fájl láncszemei, amely a nevet ellenőrizve dönti el, hogy milyen szolgáltatást biztosítson.

2-6. feladat. Miért jelez az **ls −l** négy láncszemet a recepteknél? Útmutatás:

```
$ ls −ld /usr/you
```

Miért hasznos ez az információ? □

2-7. feladat. Mi a különbség

```
$ mv vacak vacak1
```

és a

```
$ cp vacak vacak1
$ rm vacak
```

között? Útmutatás: csináljunk egy láncszemet a vacakhoz és azután próbáljuk ki! □

2-8. feladat. A **cp** aljegyzékeket nem másol, csak fájlokat egy hierarchia első szintjén. Mit csinál akkor, ha az egyik fájlargumentum egy jegyzék? Van-e értelme ennek? Elemezzük a következő három lehetőség előnyeit: a **cp** egy opciójával leereszkedünk a jegyzékekhez; egy külön parancs, az **rcp** (recursive copy) segítségével oldjuk meg a feladatot: a **cp** rekurzív módon lemásolja a jegyzéket, ha azt talál. Nézzük meg a 7. fejezetet, hogy ehhez segítséget kapjunk. Milyen egyéb programok számára lehet hasznos, hogy végig lehet menni a jegyzékfán? □

2.6. A jegyzékhierarchia

Az 1. fejezetben a **/usr/you** jegyzékből kiindulva áttekintettük a fájl rendszer hierarchiát. Most rendszerezettebben végignézzük, a fa tetejétől a gyökértől (root) kiindulva. A legfelső jegyzék a / jel.

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

A **/unix** nem más, mint a UNIX mag: amikor a rendszert elindítjuk, a **/unix** fájlt olvassuk be a lemezről a tárba és ezt indítjuk el. A folyamat két lépésben megy végbe: először a **/boot** fájlt olvassuk be, és utána ez olvassa be a **/unix** fájlt. A rendszer indításáról (bootstrap) bővebb információ a boot(8)-ban található. A / jegyzékben levő többi fájl, legalábbis a fenti listában, mind jegyzék, mindegyik valamelyest önálló része a teljes fájlrendszernek. A következő rövid hierarchiatúrán egy kicsit játszunk, felfedezzük az említett jegyzékeket. Minél jobban megismerjük a fájlrendszert, annál jobban fogjuk tudni használni. A 2.1. táblázat olyan helyeket mutat, ahol érdemes körülnézni, bár a nevek egy része rendszerfüggő.

/bin (bináris programok): mint már láttuk: ebben a jegyzékben vannak az olyan alapvető programok, mint a who vagy az ed.

/dev (devices : eszközök): a következő szakaszban ismertetjük.

etc/ (et cetera : stb.): ezzel is találkoztunk már korábban. Különböző adminisztratív fájlokat tartalmaz, mint pl. a jelszófájl, és olyan rendszerprogramok is vannak benne, mint a **/etc/getty,** ami a **/bin/login** számára inicializálja a terminálkapcsolatot. A **/etc/rc** a rendszerindítás után végrehajtandó burok parancsok fájlja. A **/etc/group** a csoportok tagjait listázza.

/lib (library : könyvtár): elsősorban a C fordító részeit tartalmazza, ilyenek a **/lib/cpp,** ami a C preprocesszora és a **/lib/libc.a,** ami a C szubrutinkönyvtára.

/tmp (temporaries : ideiglenesek): a programok végrehajtása közben keletkező rövid életű fájlokat tartalmazza. Pl. amikor az **ed** szerkesztőt elindítjuk, létrehoz egy fájlt, amelynek neve ilyen típusú: **/tmp/e00512,** ezen őrzi az éppen szerkesztett fájl másolatát, hogy ne kelljen az eredeti fájlon dolgoznia. Létrehozhatná persze ezt a fájlt a munkajegyzékünkben is, de vannak előnyei annak, hogy a **/tmp**-be teszi: pl. valószínűtlen ugyan, de már lehet esetleg egy **e00512** nevű fájlunk a jegyzékünkben; a rendszer indításakor a **/tmp** automatikusan törlődik, így a rendszer összeomlásakor nem marad felesleges fájl a jegyzékünkben, a **/tmp** általában egy gyors elérésű lemezen van.

Természetesen gondot okoz, ha egyszerre több program akar fájlokat létrehozni a **/tmp**-ben, mert zavarhatják egymást. Ezért van az **ed** ideiglenes fájljának olyan különös neve; az előállítás módja garantálja, hogy más program nem választhatja ugyanezt a nevet az ideiglenes fájljának. (Az 5. és 6. fejezetben látni fogjuk, hogyan lehet ezt megvalósítani.)

/usr (user: felhasználó): a felhasználói fájlrendszer, bár kevés köze van a rendszer tényleges felhasználóihoz. A mi gépünkön a bejelentkezési jegyzékek, a **/usr/bwk** és a **/usr/rob,** de Olvasóink gépén az 1. fejezetben elmondottak szerint a **/usr** más is lehet. Függetlenül attól, hogy saját fájljaink és a **/usr** egy aljegyzékben vannak-e, számos dolog van, amit ott találhatunk meg (bár a helyi szokások itt is különbözőek lehetnek). Ugyanúgy, mint a gyökérjegyzékben, itt is vannak **/usr/bin, /usr/lib** és **/usr/tmp** jegyzékek. Funkciójuk azonos a / jegyzékben levő névrokonaikéval, de a programjaik a rendszerre nézve kevésbé kritikusak. Pl. az **nroff** általában a **/usr/bin** jegyzékben, nem pedig a **/bin**-ben van, és a FORTRAN fordítókönyvtárai is a **/usr/lib**-ben

Érdekes jegyzékek (l. még **hier(7)**)

/	a fájlrendszer gyökere
/bin	főbb programok végrehajtható formában („binárisak")
/dev	eszközfájlok
/etc	egyéb rendszerfájlok
/etc/motd	napi bejelentkezési üzenet
/etc/passwd	jelszófájl
/lib	fontosabb könyvtárak stb.
/tmp	ideiglenes fájlok, rendszerindításkor törlődnek
/unix	az operációs rendszer végrehajtható formája
/usr	felhasználói fájlrendszer
/usr/adm	rendszeradminisztráció: elszámolási információ, stb.
/usr/bin	felhasználói binárisok: **troff** stb.
/usr/dict	szótár (**words**) és **spell(1)** támogatás
/usr/games	játékprogramok
/usr/include	C programok fejlécfájljai, pl. **math.h**
/usr/include/sys	C programok rendszer-fejlécfájljai, pl. **inode.h**
/usr/lib	C, FORTRAN stb. könyvtárak
/usr/man	on-line kézikönyv
/usr/man/man1	a kézikönyv első szakaszának lapjai
/usr/mdec	hardver-diagnosztika, rendszerindító programok stb.
/usr/news	közösségi üzenetek
/usr/pub	egyéb nyilvános dolgok, l. **ascii(7)** és **eqnchar(7)**
/usr/src	segédprogramok és könyvtárak forráskódja
/usr/src/cmd	a **/bin** és **/usr/bin** parancsainak forrása
/usr/src/lib	a szubrutinkönyvtárak forráskódja
/usr/spool	kommunikációs programok munkajegyzékei
/usr/spool/lpd	sornyomtató ideiglenes jegyzéke
/usr/spool/mail	levélszekrények
/usr/spool/uucp	az **uucp** programok munkajegyzéke
/usr/sys	az operációs rendszer magjának forrása
/usr/tmp	másodlagos ideiglenes jegyzék (ritkán használjuk)
/usr/you	bejelentkezési jegyzékünk
/usr/you/bin	személyes programjaink

vannak. Természetesen az, hogy mit tekintünk „kritikusnak", rendszerenként változik. Sok rendszerben, pl. a UNIX 7. kiadásában, minden program a **/bin**-ben van és nincs is **/usr/bin**, más rendszerekben az alkalmazás gyakoriságától függő jegyzékre bontják a **/usr/bin**-t. A **/usr** további jegyzékei: a **/usr/adm**, ami elszámolási információt és a **/usr/dict,** ami egy közepes szótárat tartalmaz (l. spell(1)). Az on-line kézikönyv a **/usr/man**-ban található (l. pl. **/usr/man/manl/spell.1).** Az **/usr/src**-ben valószínűleg megtaláljuk a rendszer forráskódját.

Érdemes a fájlrendszer, különösen a **/usr** felfedezésére időt fordítani, ezzel elképzelést alkothatunk a fájlrendszer szervezéséről, és az egyes részek elhelyezkedéséről.

2.7. Eszközök

Hierarchia túránk során átugrottuk a **/dev** jegyzéket, mert az abban levő fájlok ügyes áttekintést adnak a fájlokról általában. Ahogy a névből sejthető, a **/dev** jegyzékben az eszköz- (device) fájlok vannak.

A UNIX rendszer egyik legszebb gondolata az, ahogyan a *perifériákat* (lemezeket, mágnesszalagokat, sornyomtatókat, terminálokat stb.) kezelik. Ahelyett, hogy külön rendszerrutinokat alkalmazna mondjuk a mágnesszalag olvasására, van egy **/dev/mt0** nevű fájl (az elnevezés rendszerenként változhat). Az erre a fájlra való hivatkozások a magon belül hardverparancsokká alakulnak, így férünk hozzá a szalaghoz. Tehát ha a program a **/dev/mt0**-ról olvas, akkor a mágnesszalagegységre helyezett szalagról olvas. Pl.

```
$ cp /dev/mt0 vacak
```

a szalag tartalmát bemásolja a **vacak** nevű fájlba. A **cp** nem vesz észre semmi különöset a **/dev/mt0** hatására, neki ez is csak egy fájl, vagyis bájtok sorozata.

Az eszközfájlok olyanok, mint egy állatkert, minden teremtmény egy kicsit más, de a fájlrendszer alapgondolatai mindegyikre érvényesek. Íme a mi gépünk **/dev** jegyzékének rövidített listája:

```
$ ls -l /dev
crw--w--w-   1 root     0,  0 Sep 27 23:09 console
crw-r--r--   1 root     3,  1 Sep 27 14:37 kmem
crw-r--r--   1 root     3,  0 May  6  1981 mem
brw-rw-rw-   1 root     1, 64 Aug 24 17:41 mt0
crw-rw-rw-   1 root     3,  2 Sep 28 02:03 null
crw-rw-rw-   1 root     4, 64 Sep  9 15:42 rmt0
brw-r-----   1 root     2,  0 Sep  8 08:07 rp00
brw-r-----   1 root     2,  1 Sep 27 23:09 rp01
crw-r-----   1 root    13,  0 Apr 12  1983 rrp00
crw-r-----   1 root    13,  1 Jul 28 15:18 rrp01
crw-rw-rw-   1 root     2,  0 Jul  5 08:04 tty
crw--w--w-   1 you      1,  0 Sep 28 02:38 tty0
crw--w--w-   1 root     1,  1 Sep 27 23:09 tty1
```

```
crw--w--w-    1 root      1,  2 Sep 27 17 : 33 tty2
crw--w--w-    1 root      1,  3 Sep 27 18 : 48 tty3
$
```

Az első, amit észrevehetünk, hogy a bájtszám helyén két kis egész szám szerepel és hogy az engedélymező első karaktere mindig „b" vagy „c". Így írja ki az **ls** azt, hogy az i-bög nem „szabályos" fájlt, hanem egy eszközt ad meg. A „szabályos" fájl i-bögje a fájl tartalmát tároló lemezblokkok listáját tartalmazza. Az eszközfájl i-bögje ehelyett az eszköz belső nevét tartalmazza, amely a típusból – *karakter* (**c**) vagy *blokk* (**b**) – és egy számpárból áll, amit a fő és mellék eszközszámnak nevezünk. A lemezek és a mágnesszalagok blokkeszközök, minden más – terminál, nyomtató, telefonvonal stb. – karaktereszköz. A főszám az eszköz típusát rejti, a mellékszám az adott típus egyedeit különbözteti meg. Pl. a **/dev/tty0** és **/dev/tty1** ugyanannak a terminálvezérlőnek a két portja, ezért főszámuk azonos, de mellékszámuk különböző.

A lemezfájlok neve rendszerint mutatja az adott hardvert, pl. a **/dev/rp00** és **/dev/rp01** a rendszerhez kapcsolt DEC RP06 lemezegységről kapták a nevüket – ez van! Csak egy egység van, amelyet logikailag két fájlrendszerre osztottak. Ha volna egy második egység, a hozzá tartozó fájlok a **/dev/rp10 /dev/rp11** nevet kaphatnák. Az első jegy magát a fizikai egységet, a második az egység részét jelöli.

Olvasóink bizonyára csodálkoznak, miért van több lemezfájl ahelyett, hogy egy lenne. Történeti és karbantartási okok miatt a fájlrendszer kisebb alrendszerekre oszlik. Az alrendszerben levő fájlok a főrendszer egy jegyzékén keresztül érhetők el. Az eszközfájlok és a jegyzékek közötti megfeleltetéseket a **/etc/mount** mutatja:

```
$ /etc/mount
rp01 on /usr
$
```

A mi gépünkön a gyökérrendszer a **/dev/rp00**-án van (ezt nem írja ki a **/etc/mount**), a felhasználói fájlrendszer – a **/usr** és aljegyzékének fájljai – a **/dev/rp01**-en van.

A gyökér fájlrendszernek mindig jelen kell lennie ahhoz, hogy a rendszer működjön. A **/bin, /dev** és **/etc** mindig a gyökérrendszerhez tartozik, mert a rendszer indításakor csak gyökérrendszerben levő fájlok hozzáférhetőek, és bizonyos fájloknak – mint pl. a **/bin/sh** – már indításkor futtathatóknak kell lenniük. A rendszer ekkor minden fájlrendszer önállóságát ellenőrzi (l. icheck(8) és fsck(8)), és a gyökérrendszerhez kapcsolja azokat. Ezt a folyamatot, amely egy új lemezcsomag lemezegységre helyezésének szoftvermegfelelője, *felrakásnak* (mounting) nevezzük; ezt a tevékenységet általában csak a szuperfelhasználó végezheti. Miután a **/dev/rp01**-et felraktuk mint **/usr**-t, a felhasználói fájlrendszer fájljai ugyanúgy elérhetőek, mint ha a gyökérrendszer részei volnának.

Az átlag felhasználónak szinte mindegy, hogy melyik fájlalrendszert hova rakjuk fel, de van néhány fontos tudnivaló. Először, miután az alrendszereket felrakhatjuk vagy levehetjük, nem szabad egy másik alrendszerben levő fájlhoz láncszemet létesíteni. Pl. nem lehet a **/bin**-ben levő programokat alkalmas néven a saját **bin** jegyzékünkhöz láncolni, mert a **/usr** és a **/bin** különböző fájlalrendszerekben vannak:

```
$ ln /bin/mail /usr/you/bin/m
ln: Cross-device link
$
```

Abból is baj lehet, hogy az i-bög-számok a különböző fájlrendszerben nem egyértelműek.

Másodszor mindegyik alrendszer méretének (a fájlok számára elérhető blokkok száma) és i-bögjei számának rögzített felső korlátai vannak. Ha egy alrendszer betelik, akkor nem lehet növelni az abban levő fájlokat, amíg nem kapunk újabb helyet. A **df** (disc free space : szabad lemezterület) parancs tájékoztat arról, hogy a feltett alrendszerben mennyi szabad hely van:

```
$ df
/dev/rp00   1989
/dev/rp01   21257
$
```

A **/usr**-nek 21257 szabad blokkja van. Hogy ez sok vagy kevés, attól függ, hogyan használják a rendszert; van olyan rendszer, amely több szabad helyet igényel, van, amelyik kevesebbet. Egyébként a parancsok közül a **df** adja a legváltozatosabb formájú kimenetet. Olvasóink **df** kimenete egészen másképp nézhet ki!

Térjünk át néhány további, általánosan használható dologra! Amikor bejelentkezünk, kapunk egy terminálvonalat és így egy **/dev** fájlt, ezen keresztül kapjuk és küldjük a karaktereket. A **tty** parancs megmondja, hogy melyik terminált használjuk:

```
$ who am i
you  tty0  Sep 28 01 : 02
$ tty.
/dev/tty0
$ ls -l /dev/tty0
crw--w--w-  1 you      1, 12 Sep 28 02 : 40 /dev/tty0
$ date >/dev/tty0
Wed Sep 28 02 : 40 : 51 EDT 1983
$
```

Vegyük észre, hogy az eszköz a miénk, és hogy csak mi olvashatjuk. Másképp szólva, az általunk beírt karaktereket senki más nem olvashatja el közvetlenül. Másrészt viszont bárki írhat a terminálunkra. A **chmod** paranccsal ezt megakadályozhatjuk, ill. lehetetlenné tehetjük, hogy a **write** parancs segítségével bárki kapcsolatba lépjen velünk. A **mesg** paranccsal az üzenetek kiírását szabályozhatjuk:

```
$ mesg n                    Az üzeneteket kikapcsolja.
$ ls -l /dev/tty0
crw--------  1 you      1, 12 Sep 28 02 : 41 /dev/tty0
$ mesg y                    Az üzeneteket bekapcsolja.
$
```

Sokszor hasznos, ha a terminálra, amelyen éppen dolgozunk, névvel hivatkozha-

tunk, de kényelmetlen annak meghatározása, hogy melyik is az. A **/dev/tty** eszközként a mindenkori bejelentkezési terminált jelöli, bármilyen terminált használunk is.

```
$ date >dev/tty
Wed Sep 28 02 : 42 : 23 EDT 1983
$
```

A **/dev/tty** különösen akkor hasznos, amikor a program kapcsolatba akar lépni a felhasználóval, de a szabvány ki- és bemenete a terminál helyett fájlokhoz kapcsolódik. Az egyik olyan program, amely használja a **/dev/tty**-t, a **crypt**. A „eredeti szöveg" a szabványos bemenetről jön, a titkosított szöveg a szabványos kimenetre megy, a **crypt** a titkosító kulcsot a **/dev/tty**-ról olvassa:

```
$ crypt <eredeti-szöveg >titkosított-szöveg
Enter key:                                  Írjuk be a titkosítási kulcsot!
$
```

A **/dev/tty** használata közvetlenül nem látszik ugyan a példában, de benne van. Ha a **crypt** a szabványos bemenetről olvasná a titkosítási kulcsot, akkor az eredeti szöveg első sorát venné annak. Ehelyett a **crypt** megnyitja a **/dev/tty** fájlt, lekapcsolja az automatikus karaktervisszhangot (ezzel a titkosítási kulcs nem jelenik meg a képernyőn), és beolvassa a kulcsot. Az 5. és 6. fejezetben látni fogjuk a **/dev/tty** néhány további alkalmazását.

Előfordulhat, hogy futtatni akarjuk programunkat, de a kimenetre nincs szükségünk. Pl. már olvastuk a mai híreket és nem szeretnénk újra elolvasni. Ha a **news**-t átirányítjuk a **/dev/null** fájlba, akkor a rendszer eldobja a kimenetet:

```
$ news >/dev/null
$
```

A **/dev/null**-ba írt adat figyelmeztetés nélkül a szemétkosárba kerül, és az erről a fájlról olvasó programok azonnal fájl-vége jelet kapnak, mert **/dev/null** fájlról olvasva mindig nulla bájtokat kapunk.

A **/dev/null** egy gyakori alkalmazása, hogy a szabványos kimenet eldobásával a hibaüzeneteket tesszük láthatókká. Pl. a **time** parancs (**time(1)**) a program CPU igényéről tájékoztat. Az információ a szabványos hibafájlba íródik, így időt nyerünk, ha a terjedelmes kimenetet adó parancsok kimenetét a **/dev/null** fájlba irányítjuk:

```
$ ls -l /usr/dict/words
-r--r--r-- 1 bin   196513 Jan 20 1979 /usr/dict/words
$ time grep e /usr/dict/words >/dev/null
real        13.0
user         9.0
sys          2.7
$ time egrep e /usr/dict/words >/dev/null
real         8.0
user         3.9
sys          2.8
$
```

A **time** kimenetén megjelenő számok: a teljes idő (elapsed clock time); a program által felhasznált CPU idő; és a program futásához a mag által elhasznált CPU idő. Az **egrep** a **grep** nagyteljesítményű változata (a 4. fejezetben fogunk vele foglalkozni), amely nagy fájlokban keresés esetén körülbelül kétszer olyan gyors, mint a **grep**. Ha a **grep** vagy az **egrep** kimenetét nem küldenénk a **/dev/null** vagy egy valódi fájlba, akkor sok százezer karakter megjelenését kellene végigvárnunk, mielőtt a keresett időinformációt megkapnánk.

2-9. feladat. A kézikönyv 4. fejezetének olvasásával ismerjük meg a **/dev** további fájljait. Mi a különbség a **/dev/mt0** és a **/dev/rmt0** között? Magyarázzuk meg, milyen előnyei lehetnek annak, hogy a **/dev**-ben aljegyzékek tartoznak a lemezekhez, mágnesszalagokhoz stb! □

2-10. feladat. A nem UNIX rendszereken felírt mágnesszalagokon eltérő blokkméretek vannak (pl. 800 bájt – tíz 80-karakteres kártyakép), a **/dev/mt0** mágnesszalagegység viszont 512 bájtos blokkokat vár. Keressük meg a **dd** parancsot (l. dd(1)), nézzük meg hogyan lehet ilyen szalagot elolvasni! □

2-11. feladat. A **dev/tty** miért nem csupán egy láncszem a bejelentkezési terminál felé? Mi történne, ha ez is rw– –w– –w– engedélyekkel rendelkezne, mint a bejelentkezési terminálunk? □

2-12. feladat. Hogyan működik a **write(1)** parancs? Útmutatás: l. utmp(5). □

2-13. feladat. Miből tudjuk megmondani, hogy egy felhasználó éppen dolgozik-e a terminálon? □

Történeti és irodalmi megjegyzések

Ken Thompson „UNIX implementation" c. tanulmányának (BSTJ, 1978. július) egyik fejezete a fájlrendszerrel foglalkozik. Dennis Ritchie „The evolution of the UNIX time-sharing system" c. előadása (Symposium on Language Design and Programming Methodology, Sydney, Australia, 1979. szeptember) nagyszerű ismertetése annak, hogyan tervezték és valósították meg az eredeti PDP–7 UNIX rendszert, és hogyan fejlődött a jelenlegi formára.

A UNIX fájlrendszer átvett néhány gondolatot a MULTICS fájlrendszerből. Ez utóbbi átfogó ismertetését adja E.I Organick munkája: *The MULTICS System: An Examination of its Structure* (MIT Press, 1972). Bob Morris és Ken Thompson „Password security: a case history" c. tanulmánya élvezetes összehasonlítást ad a különböző rendszereken működő jelszómechanizmusokról, megtalálható a *UNIX Programmer's Manual* 2B kötetében.

Ugyanebben a kötetben Dennis Ritchie „On the security of UNIX" c. dolgozata elmagyarázza, hogy a rendszer biztonsága sokkal inkább múlik az adminisztráció gondosságán, mint a **crypt** és hasonló programok részletein.

3. A burok használata

A burok – a programfuttatási kéréseinket értelmező program – a UNIX felhasználók legfontosabb programja; ha nem saját kedvenc szövegszerkesztőnket használjuk, akkor többet dolgozunk a burokkal, mint bármely más programmal. Itt és az 5. fejezetben tekintélyes teret szentelünk a burok lehetőségeinek tanulmányozására. Azt szeretnénk megmutatni, hogy ha részletesen megismerkedünk a burokkal, akkor nagyon sok mindent meg tudunk oldani anélkül, hogy keményen kellene dolgoznunk, és anélkül, hogy olyan hagyományos programozási eszközöket, mint a C nyelvű programozás, igénybe kellene vennünk.

A burokkal kapcsolatos mondanivalónkat két fejezetre osztottuk. Ebben a fejezetben egy lépéssel továbbmegyünk, mint az 1. fejezetben; olyan általánosan használt, ügyes lehetőségekről lesz szó, mint a metakarakterek, az idézőjeles módok, a parancsok létrehozása, az argumentumok átadása, a burokváltozók használata, az alapvető vezérlőfolyamatok. Mindezeket ismernünk kell ahhoz, hogy a burkot alkalmazzuk. Az 5. fejezetben mélyebbre megyünk, már komoly burokrogramokat írunk, olyanokat, amelyek üzembiztosan védettek a többi felhasználó ellen. A két fejezet közötti megosztás természetesen önkényes, előbb-utóbb mindkettőt át kell tanulmányozni.

3.1. A parancssor szerkezete

Továbbhaladásunkhoz egy kicsit jobban meg kell értenünk, mi is a parancs, és hogy hogyan értelmezi azt a burok. Ez a szakasz a burok alapjairól az első fejezetben adott ismeretanyag néhány új információval kiegészített áttekintése.

A legegyszerűbb parancs egy *szó,* általában a végrehajtandó fájl megnevezése (később látunk majd másféle parancsokat is):

```
$who                                    Lefuttatja a /bin/who fájlt.
you         tty2        Sep 28 07 : 51
jpl         tty4        Sep 28 08 : 32
$
```

A parancs általában újsor-karakterrel végződik, de a ; (pontosvessző) karakter szintén *parancslezáró:*

```
$ date;
Wed Sep 28 09 : 07 : 15 EDT 1983
$ date; who
Wed Sep 28 09 : 07 : 23 EDT 1983
you          tty2       Sep 28 07 : 51
jpl          tty4       Sep 28 08 : 32
$
```

Bár a pontosvesszőt használhatjuk parancszáróként, általában semmi nem történik, amíg a *RETURN*-t meg nem nyomjuk. Megjegyezzük, hogy a burok többszörös parancsok esetén is csak egyszer írja ki a $ jelet. A $-tól eltekintve:

```
$ date; who
```

azonos azzal, mintha külön sorba írtuk volna a két parancsot. Sajátos dolog, hogy **who** nem indul el addig, amíg a **date** be nem fejeződik.

Próbáljuk a **date; who** kimenetét egy csövön keresztül küldeni:

```
$ date; who / wc
Wed Sep 28 09 : 08 : 48 EDT 1983
      2      10      60
$
```

Lehet, hogy nem ezt vártuk, hiszen a **wc** csak a **who** kimenetét kapta meg. A **who** és **wc** egy csővel összekötve egy parancsot alkot – ezt csőhálózatnak nevezzük –, amely a **date** után fut le. Amikor a burok értelmezi a parancssort, a / jelnek a pontosvesszővel szemben elsőbbsége van.

A parancsokat zárójelekkel csoportosíthatjuk:

```
$ (date; who)
Wed Sep 28 09 : 11 : 09 EDT 1983
you          tty2       Sep 28 07 : 51
jpl          tty4       Sep 28 08 : 32
$ (date; who) / wc
      3      16      89
$
```

amely a **date** és a **who** kimenetét összekapcsolja és egyetlen áradatként küldi a csőbe.

A csövön átfolyó adatokat a **tee** paranccsal meg lehet csapolni, fájlba (nem másik csőbe) lehet helyezni. Ez a parancs nem része a buroknak, de ügyes csőkezelő eszköz. A közbülső kimenetet így lehet fájlban megőrizni:

```
$ (date; who) / tee save / wc
      3      16      89                    A wc kimenete.
```

```
$ cat save
Wed Sep 28 09 : 13 : 22 EDT 1983
you          tty2       Sep 28 07 : 51
jpl          tty4       Sep 28 08 : 32
$ wc <save
       3       16       89
$
```

A **tee** a bemenetét a megnevezett fájlba vagy fájlokba és saját kimenetére másolja, tehát a **wc** ugyanazokat az adatokat kapja, mintha a csőhálózatban nem is lenne **tee**.

Újabb parancszáró karakter az & (ampersand) jel. Ugyanolyan, mint a pontosvessző vagy az újsor, de azt jelenti a buroknak, hogy ne várja meg a parancs befejeződését. Jellegzetes alkalmazásai a hosszan „háttérben" futtatott parancsok, amely közben folytathatjuk az interaktív parancsok beírását:

```
$ hosszanfutó_parancs &
5273                                   A hosszanfutó_parancs folyamatazonosítója.
$                                      A $ azonnal megjelenik.
```

Miután megismertük a parancsok csoportosításának lehetőségét, következzék a háttérfolyamatok néhány érdekes alkalmazása. A **sleep** parancs vár, amíg a megadott számú másodperc eltelik:

```
$ sleep 5
$                                      Öt másodperc eltelt, mielőtt a $ megjelent.
$ (sleep 5; date) & date
5278
Wed Sep 28 09 : 18 : 20 EDT 1983       A második date kimenete.
$ Wed Sep 28 09 : 18 : 25 EDT 1983     A $ megjelenése után 5 másodperccel a date.
```

A háttérfolyamat elkezdődik, de azonnal elalszik; ugyanakkor a második **date** parancs kiírja az időt, és a burok a $ jellel új parancsot vár. Öt másodperccel később befejeződik a **sleep** és az első **date** kiírja az új időt. Az idő múlását nehéz papíron bemutatni, úgyhogy próbáljuk ki a fenti példát! (A gép foglaltságától és egyéb részletektől függően lehet, hogy a két kiírás között nem pontosan 5 másodperc telik el.) Ennek egy egyszerű módja például a következő:

```
$ (sleep 300; echo Kész a tea) &       A tea 5 perc múlva lesz kész.
5291
$
```

Ez egy ügyes emlékeztető! (Ha a visszhangzandó sorba *ctl-g* karaktert írunk, akkor a kiíráskor a terminál csengője is megszólal.) A zárójelekre azért van szükség, mert az & magasabb prioritású, mint a pontosvessző.

A & végjel parancsokra vonatkozik, és miután a csőhálózatok parancsok, nem kell zárójeleket használni, ha csőhálózatot a háttérben futtatunk. Pl. a

```
$ pr file | lrp &
```

megszervezi a fájl sornyomtatóra írását anélkül, hogy meg kellene várnunk a parancs befejeződését. A csőhálózatot zárójelezve ugyanezt kapjuk, de többet kell gépelnünk;

```
$ (pr file | lpr) &                    Ugyanaz, mint az előbbi.
```

A legtöbb program a parancssorban elfogad argumentumokat (mint amilyen az előző példában a *file*, amely a **pr** argumentuma). Az argumentumok szóközökkel és tabulátorokkal elválasztott szavak – gyakran a parancs által feldolgozandó fájlok nevei –, de alapvetően karaktersorozatok, amiket a program tetszés szerint értelmezhet. Pl. a **pr** argumentumként a kiírandó fájlok neveit fogadja el, az **echo** értelmezés nélkül visszahangozza az argumentumot, a **grep** első argumentuma a keresendő minta, és persze a legtöbb programnak vannak opciói, amelyek – jellel kezdődő argumentumok.

A burok által értelmezett különböző speciális karakterek, a <, >, |, ; és & jelek *nem argumentumai* a burok által futtatott programoknak. Pl. a

```
$ echo Hello >vacak
```

azt mondja a buroknak, hogy az **echo**-t egyetlen argumentummal a *Hello* karaktersorozattal futtassa és a kimenetet tegye a **vacak** fájlba. A **>vacak** az **echo** parancsnak nem argumentuma; azt a burok értelmezi, az **echo** soha sem találkozik vele. Nem is kell a sor végére tenni, a

```
$ >vacak echo Hello
```

ugyanaz, mint az előző, csak kevésbé áttekinthető.

3-1. feladat. Mi a különbség a következő három parancs között:

```
$ cat file | pr
$ pr <file
$ pr file
```

(Az idők folyamán a < átirányító operátor a csövekkel szemben teret vesztett, az emberek a *cat file* | sort természetesebbnek tartják, mint a *>file* sort.) □

3.2. Metakarakterek

A burok sok karaktert különlegesen kezel. A legáltalánosabban használt ilyen karakter a * (csillag), amely a burok számára azt jelenti, hogy a katalógusban olyan fájlneveket kell keresni, amelyekben a * helyen tetszőleges karakterek állnak. Pl. a

```
$ echo *
```

az **ls** szegényes hasonmása. Az 1. fejezetben említettük, hogy minden katalógusban van . és .. nevű fájl. Ez a fájlkereséseknél kellemetlenséget okozhat, amelynek elkerülésére a fájlnév-mintakereső karakterek a ponttal kezdődő fájlneveket csak akkor nézik, ha a pontot a mintában megadjuk. Az **echo** parancs következő példái ezt szemléletesen mutatják:

```
$ ls
.profile
vacak
múlandó
$ echo *
vacak múlandó
$ echo .*
. .. .profile
$
```

Az olyan különleges tulajdonságú karaktereket, mint a *, *metakarakereknek* nevezzük. Sok ilyen van, ezeket a 3.1. táblázatban ismertetjük. Közülük néhánnyal csak az 5. fejezetben foglalkozunk.

A metakarakter különleges értelmezésére esetenként nincs szükségünk. A burok számára ezt jeleznünk kell. Ennek legegyszerűbb és legjobb módja, ha a karaktereket egyszeres (fél) idézőjelek közé tesszük:

```
$ echo '***'
***
$
```

Használhatunk idézőjeleket is (példánkban: "***"), ezen az idézőjelen belül keresi a $, `...` és a \ karaktereket. Lehetőleg csak akkor használjuk, ha az idézőjelben levő karakterláncot valahogyan fel akarjuk dolgoztatni.

A harmadik lehetőség, hogy minden egyes karakter elé, amit a buroktól védeni akarunk, fordított törtvonalat teszünk (\):

```
$ echo \*\*\*
```

Bár a *** nem látszik nagyon szép szónak, a burok nyelvén ezt is *szónak* nevezzük. A burok számára minden egyszerű karakterlánc – idézőjelek között még szóközök is lehetnek – egy egység.

Az egyikfajta idézőjel védi a másikfajtát:

```
$ echo "Don't do that!"
Don't do that!
$
```

Nem kell az egész argumentumot idézőjelekkel közrefogni:

A burok metakarakterei

>	*prog* > *fájl* közvetlen szabványos kimenet a *fájl*-ra
> >	*prog* > > *fájl* hozzáfűzött szabványos kimenet a *fájl*-ra
<	*prog* < *fájl* szabványos bemenet a *fájl*-ról
|	p_1 | p_2 a p_1 szabványos kimenetét p_2 szabványos bemenetére kapcsolja
< < *lánc*	*itt_a_dokumentum*: szabványos bemenet következik egészen a csak *lánc*-ot tartalmazó sorig
*	egyezés a fájlnevekben bármely nulla vagy több karakterből álló lánccal
?	egyezés a fájlnevekben egyetlen karakterrel
[*ccc*]	egyezés a fájlnevekben bármely *ccc* körül vett karakterrel; 0−9 vagy a−z is megengedett megadási forma
;	parancslezáró: p_1;p_2 végrehajtja először p_1-et, majd p_2-t
&	ugyanaz, mint ; de nem vár p_1 befejeződésére
`...`	futtatja a ... parancso(ka)t, a kimenetbe `...` íródik
(...)	futtatja a parancso(ka)t egy alburokban
{...}	futtatja a parancso(ka)t az aktuális burokban (ritkán használjuk)
$1, $2 stb.	$0...$9 helyére a burokfájl argumentumai kerülnek
$*var*	a *var* burokváltozó értéke
${*var*}	a *var* változó értéke; félreértést okozhat, ha szöveggel keveredik, l. még az 5.3. táblázatot
\	a *c* veszi a *c* karaktert betű szerint, *newline*-t elhagyja
'...'	veszi a ...-ot betű szerint
"..."	$ után veszi a ...-ot betű szerint, a `...`-et és a \-t értelmezi
#	ha #-el kezdődik a szó, akkor a sor többi része magyarázat (a 7. kiadásban nem így van)
var = *érték*	értéket ad a *var* változónak
p_1 && p_2	futtatja p_1-et; ha sikeres, futtatja p_2-t
p_1 || p_2	futtatja p_1-et; ha sikertelen, futtatja p_2-t

```
$ echo x'*'y
x*y
$ echo '*'A'?'
*A?
$
```

Az **echo** csak egy idézőjelek nélküli argumentumot érzékel, az idézőjelek eltűnnek, miután feladatukat ellátták. Az idézőjelben levő karakterlánc tartalmazhat újsor-karaktereket is:

```
$ echo 'jó
> reggelt'
jó
reggelt
$
```

A > egy *másodlagos prompt,* amit a burok akkor ír ki, ha a parancs befejezéséhez további bemenetet vár. Ebben a példában a burok az első sorban levő idézőjel párját várja. A másodlagos prompt karaktert a **PS2** burok változó tárolja és ízlés szerint változtatható.

Ezekben a példákban a metakaraktereket az idézőjel védi meg attól, hogy a burok megpróbálja értelmezni azokat. A

```
$ echo x*y
```

parancs minden *x*-szel kezdődő és *y*-nal végződő fájlnevet visszhangoz. Mint mindig, az **echo** most sem tud semmit sem a fájlokról és a metakarakterekről; a * értelmezését, ha szükséges, a burok végzi.

Mi történik, hogy egyik fájl sem egyezik a mintával? A burok ahelyett, hogy panaszkodna, fájlnévként a karakterláncot továbbítja, mintha az idézőjelben lett volna. Általában szerencsétlen dolog erre a viselkedésre építeni, de arra jó, hogy megtudjuk, van-e a mintának megfelelő fájl:

```
$ ls x*y
x*y not found      Üzenet a ls-től : ilyen fájl nincs.
$ >xyzzy           Létrehozunk egy xyzzy nevű fájlt.
$ ls x*y
xyzzy              Az xyzzy megfelelő a x*y mintának.
$ ls 'x*y'
x*y not found      Az ls nem értelmezi a *-ot.
$
```

A sor végén levő fordított törtvonal azt jelzi, hogy a sor folytatódik. A buroknak ezzel lehet nagyon hosszú sort megadni:

```
$ echo abc\
> def\
> ghi
abcdefghi
$
```

Figyeljük meg, hogy a \ jel utáni újsor eltűnik, szemben az idézőjelen belülivel, amelyik megmarad.

A # metakaraktert általánosan használják burok megjegyzések céljára. Ha egy burokszó # jellel kezdődik, akkor a sor további részét a burok nem értelmezi:

```
$ echo hello # there
hello
$ echo hello#there
hello#there
$
```

A # jel nincs benne az eredeti 7. kiadásban, de azóta elterjedt, és könyvünk további részeiben mi is fogjuk használni.

3-2. feladat. Magyarázzuk meg a

```
$ ls .*
```

kimenetét! □

Kalandozás az echo körül

Az **echo** a szöveg kiírása után kéretlenül egy végső újsor-karaktert ad. Ésszerűbb és világosabb tervezés lenne, ha az **echo** csak azt írná ki, amit kérünk tőle. Így egyszerűbben kapnánk a promptokat a buroktól:

```
$ tiszta_echo Adj egy parancsot:
Adj egy parancsot: $                    Nincs újsor.
```

Az újsor-karakter nélküli alapértelmezés hátránya, hogy nem ez az általános eset. Gyakrabban kell a szöveg végére az újsor, és ennél az alapértelmezésnél az új sor kiadása több gépelést igényelne:

```
$ tiszta_echo 'Hello!
> '
Hello!
$
```

Egy parancs alapértelmezése azonban a leggyakrabban alkalmazott forma, ezért az igazi **echo** automatikusan kiadja az újsort. Mi van akkor, ha nem kérjük az újsor karaktert? A 7. kiadás **echo**-jának a **−n** opciója elnyomja az újsor-karaktert:

```
$ echo −n Adj egy parancsot:
Adj egy parancsot: $      A $ ugyanebben a sorban van.
$ echo −
−                         Csak a −n hatásos.
$
```

Trükkös eset a **-n** utáni újsor visszhangoztatása:

```
$ echo -n '-n
> '
-n
$
```

Ez így elég csúnya, de működik, szerencsére nagyon ritkán fordul elő.

Az **echo** egy másik, a System V.-ben található megközelítése a C nyelv jellegű, fordított törtvonalas speciális karakterek értelmezése. Pl. \b jelentése visszalépés, és \c újsor-elhagyás (ez nincs a C nyelvben):

```
$ echo 'Adj egy parancsot: \c'        System V. változat.
Adj egy parancsot: $
```

Bár ezzel a megoldással elkerülhető a mínuszjel visszhangzása körüli zavar, vannak azonban más nehézségek. Az **echo**-t gyakran használjuk programvizsgáló eszközként, és a fordított törtvonalat nagyon sok program értelmezi. Ha ezeket az **echo** feldolgozná, csak növekedne a zavar.

Mindkét **echo**-megoldásnak vannak előnyei és hátrányai. Mi a 7. kiadás változatát használjuk (**-n**); ha Olvasóink rendszerében másképp van, akkor néhány programunkban apróbb változtatásokat kell végrehajtaniuk.

Egy másik érdekes kérdés, hogy mit csináljon az **echo**, ha *nem* kap argumentumot – üres sort írjon vagy semmit sem? A ma használt összes változat ilyen esetben üres sort ír, de régebben ez nem volt ennyire egyértelmű. A kérdéskör valaha volt nagy vitáit Doug McIlroy következő „tanulmánya" is mutatja:

A UNIX és a visszhang

Hol volt, hol nem volt, még az Óperenciás tengeren is túl, New Jersey földjén, élt egyszer egy tündérszép hajadon, a UNIX, aki olyan szépséges volt, hogy messzi földről csodájára jártak a tudósok. Tiszta ártatlansága úgy elkápráztatta őket, hogy mind feleségül akarta venni; az egyik hamvas bájáért, a másik csiszolt modoráért, a harmadik pedig azért, mert olyan nehéz feladatokat is meg tudott oldani, amilyeneket még a sokkal gazdagabb vidékeken is csak ritkán tudtak. UNIX olyan jószívű, kedves természetű volt, hogy mindegyik kérője kívánságait teljesítette, csak a legelviselhetetlenebbül gazdagokat utasította el. Hamarosan rengeteg sarjuk lett, mind szépen fejlődtek, és eljutottak még a világ végéig is.

A Természet csak mosolygott, és lelkesebben válaszolgatott UNIX-nak, mint a többi halandó teremtménynek. Az alázatos népség, amely nemigen ismerte a finomabb szokásokat, csodálta pontos, kristálytiszta visszhangját. Alig hitte, hogy ugyanazok a sziklák és erdők visszhangozzák UNIX-ot, amelyek az ő vadonba kiáltott hangjukat olyan torzan adják vissza. A készséges UNIX tökéletes visszhanggal szolgált, bármit is kérdeztek tőle.

Amikor az egyik türelmetlen kérő ezt kérte UNIX-tól: „Visszhangozz *semmi*-t!", Ő engedelmesen kinyitotta a száját, *semmi*-t visszhangzott, majd becsukta. „Hogy képzeled, hogy így kinyitod a szájadat?" – kérdezte az ifjú. „Mostantól kezdve sose nyisd ki, ha *semmi*-t kell visszhangoznod!" És UNIX engedelmeskedett.

„Legyél tökéletes akkor is, amikor *semmi*-t visszhangzol!" – szólt egy érzékeny ifjú. „Csukott szájjal nem adhatsz tökéletes visszhangot!" UNIX egyiküket sem akarta megbántani, ezért azt találta ki, hogy különböző *semmi*-t fog mondani a türelmetlen ifjúnak és az érzékeny ifjúnak. Az érzékeny ifjúnak a *semmi*-t \n-nek nevezte.

Most viszont, amikor \n-t mondott, igazából nem *semmi*-t mondott, ezért kétszer kellett kinyitnia a száját: egyszer, hogy \n-t mondjon, és egyszer, hogy *semmi*-t mondjon. Így aztán ez sem tetszett az

érzékeny ifjúnak, aki így szólt legott: „A \n tökéletes *semmi*-nek hangzik, de a másik elrontja az egészet. Azt akarom, hogy vond vissza az egyiket." Ezért UNIX, aki a világ minden kincséért sem bántott volna meg senkit, kitalálta, hogy visszavon néhány visszhangot, és ezeket \c-nek nevezte. Az érzékeny ifjú most már a *semmi* tökéletes visszhangját hallhatta, ha egyszerre kérte a \n-et és a \c-t. Azt mesélik azonban, hogy a sok jelöléstől csömört kapott, bele is halt, és azóta már mindig csak a tökéletes *semmi*-t hallja.

3-3. feladat. Mit csinálnak az itt következő **grep** parancsok? Győződjünk meg arról, hogy valóban azt csinálják!

```
grep \$          grep \\
grep \\$         grep \\\\
grep \\\$        grep "\$"
grep '\$'        grep '"$'
grep '\'$'       grep "$"
```

Segíti az ellenőrzést, ha olyan fájlt csinálunk, amely magukat ezeket a parancsokat tartalmazza. □

3-4. feladat. Hogyan adjuk meg a **grep**-nek, hogy mínuszjellel kezdődő mintát keressen? Miért nem segít, ha az argumentumot idézőjelbe tesszük? Útmutatás: gondoljunk a **−e** opcióra! □

3-5. feladat. Kiírja-e az összes jegyzéket a következő parancs:

```
$ echo */*
```

Milyen sorrendben jelennek meg a nevek? □

3-6. feladat. (Ravasz kérdés.) Hogyan tehetünk / jelet a fájlnévbe (úgy, hogy ne válassza szét az ösvény komponenseit)? □

3-7. feladat. Mi történik a

```
$ cat x y >y
```

és a

```
$ cat x >>x
```

hatására? Gondolkodjunk, mielőtt továbblépünk! □

3-8. feladat. Miért nem figyelmeztet az **rm** parancs arra, hogy a

```
$ rm *
```

beírásával az összes fájlunkat töröljük? □

3.3. Új parancsok létrehozása

Ideje, hogy rátérjünk arra, amit még az első fejezetben ígértünk: hogyan lehet új parancsokat létrehozni a már meglevőekből?

A parancsok egy-egy sorozatát kényelmesen megismételhetjük, ha egy „új" parancsot csinálunk belőle, nevet adunk neki, és ugyanúgy használjuk, mint a rendes parancsokat. Példaként tegyük fel, hogy az 1. fejezetben említett

```
$ who | wc -l
```

csőhálózattal gyakran meg kívánjuk számlálni a felhasználókat. Csináljunk erre egy **nu** nevű új programot!

Első lépésként létrehozunk egy közönséges fájlt, amelynek tartalma: *'who | wc -l'*.

Használhatjuk ehhez szövegszerkesztőnket, de létrehozhatjuk az alábbi elmés módon is:

```
$ echo 'who | wc -l' >nu
```

(Az idézőjel nélkül mi kerülne a **nu**-ba?)

Ahogy az első fejezetben mondtuk, a burok ugyanolyan program, mint a szövegszerkesztő, a **who** vagy a **wc** és a neve **sh**. Minthogy ez egy program, futtatható, és a bemenete átirányítható. Futtassuk tehát a burkot úgy, hogy a bemenete ne a terminálról, hanem a **nu** fájlból jöjjön:

```
$ who
you     tty2   Sep 28 07 : 51
rhh     tty4   Sep 28 10 : 02
moh     tty5   Sep 28 09 : 38
ava     tty6   Sep 28 10 : 17
$ cat nu
who | wc -l
$ sh <nu
        4
$
```

A kimenet ezután ugyanaz, mintha a terminálon a *who | wo -l* parancsokat írtuk volna be.

Mint sok más program, a burok is az argumentumként megadott fájlból veszi a bemenetét, tehát

```
$ sh nu
```

ugyanezt az eredményt adja. Kényelmetlen, hogy mindig be kell írnunk az **sh** parancsot; hosszadalmas és különbözőséget okoz. Másképp kell futtatni pl. a C nyelven írt programokat, mint azokat, amelyek burok segítségével összekapcsolt programok-

ból állnak.* Ennek a kellemetlenségnek az elkerülése érdekében, ha egy fájl végrehajtható és szöveget tartalmaz, akkor a burok azt feltételezi, hogy a fájl burokparancsokat tartalmaz. Az ilyen fájlt *burokfájlnak* nevezzük. Ezért annyit kell csak tennünk, hogy a **nu** fájlra egyszer végrehajtás engedélyt adunk:

```
$ chmod +x nu
```

és ezek után bármikor hívhatjuk így:

```
$ nu
```

Ettől kezdve a **nu** felhasználói nem tudják megmondani, hogy ezt az új parancsot ilyen egyszerűen hoztuk létre.

A burok úgy futtatja a **nu** parancsot, hogy egy új burokfolyamatot hoz létre. Pontosan ugyanúgy, mintha ezt írtuk volna:

```
$ sh nu
```

Ezt a gyerekburkot *alburoknak* (sub-shell) nevezzük – ez az aktuális burkunk által hívott burokfolyamat. Az *sh nu* nem ugyanaz, mint az *sh <nu*, mert szabványos bemenete a terminálhoz kapcsolódik.

Most a **nu** csak akkor működik, ha munkajegyzékünkben található (feltéve persze, hogy munkajegyzékünk benne van a PATH-ban, a továbbiakban ezt mindig feltételezzük). Ha függetlenül attól, hogy éppen milyen jegyzékben vagyunk, mindig elérhetővé szeretnénk tenni a **nu** parancsot, akkor tegyük a saját **bin** jegyzékünkbe, és a keresési úthoz tegyük hozzá a **/usr/you/bin** nevet:

```
$ pwd
/usr/you
$ mkdir bin                    Hozzunk létre bin jegyzéket, ha még nem volt!
$ echo $PATH                   Ellenőrizzük a PATH-ot a biztonság kedvéért!
:/usr/you/bin:/bin:/usr/bin    Ennek kell lennie!
$ mv nu bin                    Beépítjük a nu-t.
$ ls nu
nu not found                   Eltűnt a munkajegyzékből.
$ nu
      4                        A burok azonban megtalálja.
$
```

Természetesen nem kell ezt minden bejelentkezéskor újra beírni. A **.profile** segítségével ehhez a **PATH**-ot kell jól beállítani.

További egyszerű parancsok létrehozásával tetszésünk szerint módosíthatjuk a rendszert. Íme néhány parancs, amit kényelmesnek találtunk:

• **`cs**, amely bűvös karakterek sorozatát a képernyőre visszhangozva (általában egyszerűen 24 újsor-karaktert) törli a képernyőt.

* Vigyázat, ezeket a legtöbb operációs rendszer különbözőnek tekinti!

• **what,** ami a **who** és a **ps** **−a** parancsokat futtatja. Megmondja, kik vannak bejelentkezve és mit csinálnak.

• **where,** amely kiírja az éppen használt UNIX rendszer azonosítónevét − előnyös, ha rendszeresen többfélét használunk. (A **PS1** beállítása hasonló célt szolgál.)

3-9. feladat. Nézzük meg a **/bin** és a **/usr/bin** jegyzékekben, hogy hány olyan parancs van, amely valójában burokfájl! Meg tudjuk ezt nézni egy paranccsal? Útmutatás: l. file(1)! Mennyire pontos a fájlok hossza szerinti sejtés? □

3.4. Parancsargumentumok és paraméterek

A **nu** parancs önmagában is tökéletes, a legtöbb burokprogram azonban argumentumokat is értelmez, így a program futásakor pl. fájlnevek és opciók adhatók meg.

Tegyük fel, hogy olyan **cx** nevű programot szeretnénk alkotni, amely egy fájlra végrehajtási engedélyt ad, vagyis

```
$ cx nu
```

legyen a

```
$ chmod +x nu
```

rövidítése. Már tudunk annyit, hogy ezt megalkossuk. Kell hozzá egy **cx** nevű fájl, aminek a tartalma:

```
chmod +x fájlnév
```

Az egyetlen új dolog, amit még nem tudunk az, hogy hogyan mondhatjuk meg a **cx** programnak a fájlnak a nevét, hiszen az a **cx** minden futtatásakor más és más lesz.

Amikor a burok egy parancsfájlt futtat, akkor a *$1* helyére mindig az első argumentumot, a *$2* helyére a második argumentumot teszi, és így tovább *$9*-ig. Tehát, ha a **cx** fájl tartalma

```
chmod +x $1
```

akkor, ha a

```
$ cx nu
```

parancsot futtatjuk, az alburok a *$1* helyére az első argumentumot, a **nu**-t teszi.

Nézzük a lépések teljes sorozatát:

```
$ echo 'chmod +x $1' >cx        Először is létrehozzuk a cx-et.
$ sh cx cx                      cx-et futtathatóvá tesszük.
$ echo echo Itt vagyok! >hello  Tesztprogramot készítünk.
$ hello                         Kipróbáljuk.
hello: cannot execute
$ cx hello                      Végrehajthatóvá tesszük.
$ hello                         Újra kipróbáljuk.
Itt vagyok!                     Működik!
$ mv cx /usr/you/bin            Beépítjük a cx-et!
$ rm hello                      Törlünk.
$
```

Megjegyezzük, hogy

```
$ sh cx cx
```

pontosan ugyanazt teszi, mint amit a burok végrehajtható **cx** esetén

```
$ cx cx
```

hatására automatikusan tett volna.

Mi történik, ha több argumentumot akarunk kezelni, pl. olyan **cx** programot csinálunk, amely egyszerre több fájlt kezel? Egy gyenge megoldás, hogy a burokprogramnak kilenc argumentumot adunk, így:

```
$ chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

(Csak *$9*-ig működhet, mert a *$10* értelme: a *$1* első argumentumot *0* követi!) Ha ennek a burokfájlnak kilencnél kevesebb argumentumot adunk meg, akkor a hiányzók nulla karakterláncok lesznek; az alburok csak a valóban megadott argumentumokat adja át a **chmod**-nak. Ez tehát így egy működő, de nyilvánvalóan ügyetlen megoldás, és kilencnél több argumentumra már nem is használható.

A feladat igazi megoldása: a burok számára a *$** rövidítés az összes argumentumot jelenti. Ezek után a **cx** definíciója a következő:

```
chmod +x $*
```

ami az argumentumok számától függetlenül működik. A *$** felhasználásával ügyes burokfájlokat alkothatunk, mint az **lc** vagy **m:**

```
$ cd /usr/you/bin
$ cat lc
#lc: megszámolja a fájlokban levő sorokat
wc -l $*
$ cat m
#m: a levélírás egy tömör módja
mail $*
$
```

Mindkettő használható argumentumok nélkül is. Ha nincs argumentum, akkor a *$** nulla lesz, és a **wc**, ill. **mail** nem kap argumentumot. A parancs is argumentummal vagy anélkül hívható:

```
$ lc /usr/you/bin/*
   1 /usr/you/bin/cx
   2 /usr/you/bin/lc
   2 /usr/you/bin/m
   1 /usr/you/bin/nu
   2 /usr/you/bin/what
   1 /usr/you/bin/where
   9 total
$ ls /usr/you/bin | lc
   6
$
```

Ezek és az ebben a fejezetben található egyéb parancsok *személyes* programok, olyanok, amelyeket magunk írunk, saját **bin** jegyzékünkbe teszünk. Nem érdemes nagydobra verni őket, mert egyéni ízlésünktől túlságosan függenek. Az 5. fejezetben meg fogjuk mutatni, hogyan lehet mindenki által használható burokprogramokat írni.

A burokfájl argumentumai nemcsak fájlnevek lehetnek. Pl. tegyük fel, hogy saját telefonszámjegyzékünkben keresünk. Legyen egy **/usr/you/lib/telefon** nevű fájlunk és legyen annak tartalma a következőhöz hasonló:

```
mesemondó_száma 212-838   Nem valódi hívószámok!
lelki_segély_száma 212-200
sánta_száma 212-636
kis_János_sógora 212-141
```

Használjuk keresésre a **grep** parancsot! (Az ilyen személyes adatbázisok tárolására megfelelő hely a saját **lib** jegyzékünk.) Miután a **grep** nem nézi az információ formáját, kereshetünk vele neveket, címeket, telefonszámokat vagy bármit, ami tetszik. Csináljunk segédprogramot, nevezzük azt **411**-nek (annak emlékére, hogy a szerzők lakhelyén ennyivel kezdődnek a telefonszámok):

```
$ echo 'grep $* /usr/you/lib/telefon' >411
$ cx 411
$ 411 mese
mesemondó_száma 212-838
$ 411 száma
mesemondó_száma 212-838
lelki_segély_száma 212-200
sánta_száma 212-636
$ 411 'János sógora'
grep: can't open sógora        Valami nem jó!
$
```

Az utolsó példa egy lehetséges hibát mutat be: bár *jánosi sógora* a **411** számára egyetlen argumentum, de szóközt tartalmaz, és már nincs idézőjelben, így a **411** parancsot értelmező alburok két argumentummá alakítva adja a **grep** parancsnak, mintha ezt írtuk volna:

```
$ grep jánosi sógora /usr/you/telefon
```

Ez nyilvánvalóan rossz. A buroknak az idézőjel-kezelése segít. Bár ami fél idézőjelek között ('...') van, az sérthetetlen, a burok az idézőjelek között ("...") *$*, \ jeleket és fordított fél idézőjeleket (` ... `) keres. Alakítsuk át a **411** parancsot a következő formára:

```
grep "$*" /usr/you/lib/telefon
```

A *$** helyére bekerülnek az argumentumok, de azokat a **grep** egyetlen argumentumként kapja meg, még akkor is, ha van benne szóközkarakter.

```
$ 411 jánosi sógora
kis jánosi sógora 212-141
$
```

Egyébként a **grep**-et (és így a **411**-et is) a **−y** opcióval kiegészítve a kisbetűk és nagybetűk megkülönböztetését megszüntethetjük:

```
$ grep −y minta...
```

A **−y** hatására a minta kisbetűi megegyeznek a bemenet nagybetűivel is. (Ez a lehetőség a 7. kiadás **grep**-jében megvan, némely másikból hiányzik.)

A parancsargumentumokkal kapcsolatos néhány további finomságot az 5. fejezetben fogunk tárgyalni, de egyet már most érdemes megjegyezni. A *$0* a végrehajtás alatt álló program neve − a **cx**-ben a *$0* maga a **cx**. A *$0* nevezetes alkalmazásai a kimenetet több oszlopban kiíró **2, 3, 4,** ..., nevű programok:

```
$ who | 2
drh     tty0    Sep 28 21 : 23      cvw     tty5    Sep 28 21 : 09
dmr     tty6    Sep 28 22 : 10      scj     tty7    Sep 28 22 : 11
you     tty9    Sep 28 23 : 00      jlb     ttyb    Sep 28 19 : 58
$
```

A **2, 3,** ..., megvalósítása azonos, ugyanannak a fájlnak a láncszemei.

```
$ ln 2 3; ln 2 4; ln 2 5; ln 2 6
$ ls −li [1−9]
16722 −rwxrwxrwx 5 you      51 Sep 28 23 : 21 2
16722 −rwxrwxrwx 5 you      51 Sep 28 23 : 21 3
```

```
16722 -rwxrwxrwx 5 you        51 Sep 28 23 : 21 4
16722 -rwxrwxrwx 5 you        51 Sep 28 23 : 21 5
16722 -rwxrwxrwx 5 you        51 Sep 28 23 : 21 6
$ ls /usr/you/bin / 5
2          3          4          411        5
6          cx         lc         m          nu
what       where
$ cat 5
# 2, 3, ... : print in n columns
pr -$0 -t -l1 $*
$
```

A **−t** opció megszünteti a fejlécet a papíron, a **−ln** opció beállítja a lap hosszát n sorra. A program neve lesz a **pr** oszlopszám argumentuma. A kimenet egyszerre egy sort nyomtat, annyi oszlopban, amennyit a *$0*-val megadtunk.

3.5. Programkimenet mint argumentum

A burokfájlon belüli parancsargumentumok után foglalkozzunk most az argumentumok előállításával! Az argumentumokat legegyszerűbben az előző részben tárgyalt * típusú metakarakterekkel lehet előállítani. Egy másik jó módszer: programot futtatni argumentumként. Bármely program kimenetét elhelyezhetjük a parancssorban, ha hívását fordított fél idézőjelek közé (`...`) tesszük:

```
$ echo A pontos idő `date`.
A pontos idő Thu Sep 29 00 : 02 : 15 EDT 1983.
$
```

Egy kis változtatással látni fogjuk, hogy a fordított fél idézőjelek (`...`) az idézőjelek között ("...") is érvényesek:

```
$ echo " A pontos
> idő `date` ."
A pontos
idő Thu Sep 29 00 : 03 : 07 EDT 1983.
$
```

Egy másik példa: levelet szeretnénk küldeni azoknak, akiknek bejelentkező neve a **címlista** nevű fájlunkban van. Nem szép megoldás, a címlistafájlt beszerkesztjük egy megfelelő **mail** parancsba és ezt adjuk a buroknak. Sokkal egyszerűbb a következő:

```
$ mail `cat címlista` <levél
```

Ennek hatására a **cat** sorra előveszi a felhasználók nevét, amelyek így válnak egymás után a **mail** argumentumává. (Amikor argumentumként fordított fél idézőjelben

kimenetet adunk meg, akkor a burok az újsor-karaktert nem parancssor-lezáróként, hanem szóelválasztóként kezeli. Ezt részletesen az 5. fejezetben ismertetjük.) A fordított idézőjelek használatának egyszerűsége tette feleslegessé a **mail** parancslista opciójának kialakítását.

Egy kicsit más megközelítésben a címlistafájlt a nevek egyszerű listájából olyan programmá alakítjuk át, amely kiírja a neveket:

```
$ cat címlista                    Új változat.
echo don whr ejs mb
$ cx címlista
$ címlista
don whr ejs mb
$
```

Ezután a listán levőknek így postázhatjuk a levelet:

```
$ mail `címlista` <levél
```

Még egy program hozzáadásával interaktívan módosíthatjuk a listát. Ez a program a **pick** :

```
$ pick argumentumok ...
```

amely egyenként veszi az argumentumokat, és mindegyik után választ vár. Kimenetén azok az argumentumok jelennek meg, amelyeket a *y* (yes : igen) válasszal kiválasztottunk; egyéb válasz esetén nem jelenik meg az argumentum. Pl.:

```
$ pr `pick *.c` | lpr
```

előszedi a *.c*-re végződő fájlneveket, amelyeket azután a **pr** és **lpr** kiír. (A **pick** nem része ugyan a 7. kiadásnak, de olyan hasznos és egyszerű, hogy az 5. és 6. fejezetbe néhány változatát felvettük.) Tegyük fel, hogy a címlista második változatát készítettük el. Ekkor

```
$ mail `pick \ `címlista \ `` <levél
don? y
whr?
ejs?
mb? y
$
```

csak **don**-nak és **mb**-nek küldi el a levelet. Figyeljük meg az egymásba ágyazott fordított fél idézőjeleket. A fordított törtvonal megakadályozza a belső fordított idézőjel (`...`) értelmezését, *amíg a külsőé folyik.*

3-10. feladat. Mi történik, ha a következő parancssorból kihagyjuk a fordított törtvonalakat?

```
$ echo `echo \ `date\ ``
```

□

3-11. feladat. Próbáljuk ki a következő parancsot, és magyarázzuk meg az eredményt! □

```
$ `date`
```

3-12. feladat.

```
$ grep -l minta fájlnevek
```

kiírja azokat a fájlneveket, amelyekben előfordul a *minta* szó, de más kimenetet nem ad. Próbáljunk ki néhány

```
$ parancs `grep -l minta fájlnevek`
```

változatot! □

3.6. Burokváltozók

A buroknak, mint a legtöbb programnyelvnek, vannak változói. Ezeket néha *paramétereknek* nevezzük. Az olyan karakterláncok, mint a *$1*, *pozicionális paraméterek* – egy burokfájl argumentumait tartalmazó változók. A számjegy a parancssorban elfoglalt helyet jelzi. Láttunk már más burokváltozókat is: a **PATH** a parancskiválasztás katalógusának listája; a **HOME** a bejelentkezési katalógusunk stb. A szabályos nyelvek változóival ellentétben az argumentumváltozók értéke nem változik: a **PATH** változó értéke $PATH, viszont nincs olyan **1** nevű változó, amelynek az értéke $1 lenne. $1 nem más, mint az első argumentum tömör jelölése.

A pozicionális paraméterektől eltekintve a burokváltozók létrehozhatóak, elérhetőek és módosíthatóak. Pl. a

```
$ PATH=:/bin:/usr/bin
```

értékadással megváltoztatjuk a keresési utat. Az egyenlőségjel előtt és után nem lehet szóköz. Hozzárendelni csak egy szót lehet, ami azt jelenti, hogy idézőjelbe kell tenni a nem értelmezendő burok-metakaraktereket. A változó értékét úgy kapjuk meg, hogy a név elé dollárjelet írunk:

```
$ PATH=$PATH:/usr/games
$ echo $PATH
:/usr/you/bin:/usr/bin:/usr/games
$ PATH=:/usr/you/bin:/bin:/usr/bin      Helyreállítja.
$
```

Nem csak különleges burokváltozók vannak. Mi is létrehozhatunk új változókat úgy, hogy értéket adunk nekik; megkülönböztetésül a különleges jelentésű változókat szokás nagybetűkkel, az egyszerű neveket pedig kisbetűkkel írni. Gyakran használjuk a változókat hosszú karakterláncok, mint pl. ösvénynevek jelölésére:

```
$ pwd
/usr/you/bin
$ dir=`pwd`                Megjegyezzük, hol voltunk.
$ cd /usr/mary/bin         Máshova megyünk.
$ ln $dir/cx .             A fájlnévben használjuk a változót.
$ ...                      Dolgozunk egy ideig.
$ cd $dir                  Visszatérünk.
$ pwd
/usr/you/bin
$
```

A burokba beépített **set** parancs az összes megadott változónk értékét kiírja. Ha csak egyre vagy kettőre vagyunk kíváncsiak, célszerűbb az **echo**-t használni:

```
$ set
HOME=/usr/you
IFS=
PATH=:/usr/you/bin:/bin:/usr/bin
PS1=$
PS2=>
dir=/usr/you/bin
$ echo $dir
/usr/you/bin
$
```

A változó értéke az őt létrehozó burokhoz tartozik, nem adódik át automatikusan a burok gyermekeinek.

```
$ x=Hello          Létrehozzuk az x-et.
$ sh               Új burok
$ echo $x
                   Csak újsor: x az alburok számára ismeretlen.
$ ctl-d            Kilépünk ebből a burokból.
$                  Vissza az eredetihez.
$ echo $x
Hello              x újra ismert.
$
```

Ez azt jelenti, hogy a burokfájl nem változtathatja meg egy változó értékét, mert a burokfájlt egy alburok futtatja:

```
$ echo 'x=" Jó éjszakát"     Csináljunk egy kétsoros burokfájlt,
> echo $x' >setx             amely beállítja és kiírja x értékét!
$ cat setx
```

```
x="Jó éjszakát"
echo $x
$ echo $x
Hello                    x az eredeti burokban levő Hello.
$ sh setx
Jó éjszakát              x az alburokban levő Jó éjszakát.
$ echo $x
Hello                    De ebben a burokban még mindig Hello van!
$
```

Esetenként a burokváltozókat jó lenne burokfájl segítségével változtatni. Nyilvánvaló példa erre, ha egy fájllal újabb jegyzéket akarunk a **PATH**-hoz hozzátenni. A burok ezt a . (pont) paranccsal támogatja, amely egy fájl parancsait az aktuális burokban, nem pedig egy alburokban futtatja. Ezt eredetileg arra találták ki, hogy a **.profile** fájlt bejelentkezés nélkül újból le lehessen futtatni, de van más alkalmazása is:

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games    A /usr/games-t felfűzzük a PATH-ra.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games
$
```

A . parancs a fájlját a **PATH**-mechanizmussal keresi, ezért helyezhettük azt a saját **bin** jegyzékünkbe.

Amikor egy fájlt a . paranccsal futtatunk, az csak látszólag olyan, mint ha egy burokfájlt futtatnánk. A fájl a szó szokásos értelmében nem kerül „végrehajtásra". Ehelyett a benne levő parancsokat a burok úgy értelmezi, mintha interaktívan írtuk volna be azokat – a burok bemenetére ideiglenesen a fájlból érkeznek az adatok. Miután ez olvasás és nem futtatás, nem kell hozzá végrehajtás-engedély. A másik különbség, hogy a fájl nem kap parancssor-argumentumokat; **$1, $2** és a többi üres marad. Jó lenne, ha lehetne argumentumokat átadni, de sajnos nem megy.

A másik lehetőség az alburok egy változója értékének beállítására, hogy a parancssorban a parancs előtt explicite megadjuk az értéket:

```
$ echo 'echo $x' >echox
$ cx echox
$ echo $x
Hello                    Mint korábban.
$ echox

                         Az alburokban x-nek nincs értéke.
$ x=Hi echox
Hi                       Az x értékét átadta az alburoknak.
$
```

(Eredetileg a parancssorba bárhova lehetett értékadást írni, de ez zavarta a **dd(1)** parancsot.)

A változó értékének állandó megváltoztatására a . parancsot, az ideiglenes változtatásra a parancssorban való megadást kell használni. Példaként tekintsük újra a parancskeresést a **/usr/games**-ben egy olyan jegyzékkel, amely nincs benne a **PATH** változónkban.

```
$ ls /usr/games | grep fort
fortune                                  Egy szerencsés parancs.
$ fortune
fortune: not found
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin              /usr/games nincs a PATH-ban
$ PATH=/usr/games/fortune
Ring the bell; close the book; quench the candle.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin              A PATH változatlan.
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games                    A games parancs még ott van.
$ . games
$ fortune
Premature optimization is the root of all evil – Knuth
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games   A PATH most megváltozott.
$
```

A két mechanizmust egyesíthetjük egy burokfájlban. Kicsit eltérő **games** parancsal a **PATH** megváltoztatása nélkül egy játékot futtathatunk, vagy beállíthatjuk **PATH**-ot úgy, hogy állandóan tartalmazza a **/usr/games**-t:

```
$ cat /usr/you/bin/games
PATH=$PATH:/usr/games $*                 Figyeljük meg a $*-ot!
$ cx /usr/you/bin/games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin              Nincs benne a /usr/games.
$ games fortune
I'd give my right arm to be ambidextrous.
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin              Még mindig nincs benne.
$ . games
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin:/usr/games   Most már benne van.
$ fortune
He who hesitates is sometimes saved
$
```

A **games** első hívása a burokfájlt egy alburokban futtatta, ahol a **PATH**-ot ideiglenesen úgy módosítottuk, hogy tartalmazza a **/usr/games**-t. A második példa pedig az aktuális burokban értelmezte a fájlt, az üres *$** karakterlánc miatt a sorban nem volt

parancs, csak a **PATH**-ot módosítottuk. A **games** e két trükkös alkalmazása kényelmes, természetesen használható lehetőséget ad.

Egy változó értéket az alburok számára is elérhetővé tehetjük a burok **export** parancsával. (Gondolkodjunk el azon, hogy miért nem lehet a változó értékét az alburokból exportálni a szülőburokba!) Íme egy régi példánk, most **export** paranccsal megadott változóval:

```
$ x=Hello
$ export x
$ sh                    Új burok.
$ echo $x
Hello                   Az alburokban x ismert.
$ x='Jó éjszakát'       Megváltoztatjuk az értékét.
$ echo $x
Jó éjszakát
$ ctl-d                 Kilépünk ebből a burokból.
$                       Vissza az eredeti burokba.
$ echo $x
Hello                   Az x még mindig Hello.
$
```

Az **export** parancs szemantikája szövevényes, de a mindennapos gyakorlatban elfogadható a következő szabály: rövid időre beállított ideiglenes változókra ne használjunk **export** parancsot, de mindig használjuk azokra a változókra, amelyeket minden burokban és alburokban (beleértve pl. az **ed !** parancsával indított burkot is) azonos értéken kívánunk szerepeltetni. Tehát a burok különleges változóit, mint a **PATH** és **HOME** változókat, mindig **export** paranccsal adjuk meg.

3-13. feladat. Miért építjük be a munkakatalógust mindig a **PATH**-ba? Hova kell beépíteni? □

3.7. Még többet az átirányításról

A szabványos hibát azért találták ki, hogy a hibaüzenet mindig megjelenjék a terminálon:

```
$ diff file1, file2 >diff.out
diff: file2: No such file or directory
$
```

Kívánatos, hogy a hibaüzenetek így működjenek – nagyon megtévesztő lenne, ha eltűnnének a **diff.out**-ban, hiszen a hibás **diff** parancsról is azt hinnénk, hogy az jól működik.

Minden program az indításakor alapértelmezés szerint három fájlt hoz létre, amelyeket fájlleírónak (file descriptor) nevezett kis egész számokkal jellemzünk (a 7.

fejezetben még visszatérünk erre). A **0** szabványos bemenetet és az **1** szabványos kimenetet, amelyet már megismertünk, gyakran átirányítjuk fájlokra vagy csövekre. Az utolsóként említett **2** szabványos hibakimenet általában a terminálra kerül.

Néha a programok akkor is adnak szabványos hibaüzenetet, ha jól működnek. Az egyik legismertebb példa erre a **time** program, ami lefuttat egy parancsot, majd a szabvány hibaüzenetként jelzi, hogy mennyi ideig futott:

```
$ time wc sz3.1
  931     4288     22691     sz3.1

real        1.0
user        0.4
sys         0.4
$ time wc sz3.1 >wc.out

real        2.0
user        0.4
sys         0.3
$ time wc sz3.1 >wc.out 2>time.out
$ cat time.out

real        1.0
user        0.4
sys         0.3
$
```

A *2>fájlnév* szerkezettel (a *2* és a > jel között nem lehet szóköz) a szabványos hibaüzeneteket a fájlba irányítjuk; szintaktikusan ugyan esetlen a megoldás, de végrehajtja a feladatot. (Az ilyen rövid parancsoknál a **time** által kiírt idő nem pontos, de több hosszú teszt esetén az értékek megbízhatóak, ha ezeket a további elemzéshez tároljuk. Nézzük meg pl. a 8.1. táblázatot!)

Két kimenő áradatot össze lehet válogatni:

```
$ time wc sz3.1 >wc.out 2>&1
$ cat wc.out
    931     4288     22691     sz3.1
real        1.0
user        0.4
sys         0.3
$
```

A *2>&1* jelölés a burok számára azt jelenti, hogy a szabványos hibaüzenetet ugyanarra az áradatra tegye, mint a szabványos kimenetet. Az *&* jelnek itt nincs semmi különösebb jelentése, egyszerűen csak ki kell írni. Az *1>&2* alkalmazásával a szabványos kimenetet hozzátehetjük a szabványos hibaüzenethez:

```
echo ... 1>&2
```

Így a szabványos hibaüzenet kimenetén fog megjelenni a szabványos kimenet is. Burokfájlok esetén ez megakadályozza, hogy az üzenetek véletlenül eltűnjenek egy csőben vagy egy fájlban.

A burok lehetőséget ad arra, hogy egy parancs szabványos bemenetét külön fájl helyett a paranccsal együtt adjuk meg. Így a burokfájl teljesen öntartalmazó. A **411** jegyzékinformációs programunk így írható:

```
$ cat 411
grep "$*" <<End
mesemondó_száma     212-838
lelki_segély_száma  212-200
sánta_száma         212-636
kis_János_sógora    212-141
End
$
```

A burok nyelvében ezt **itt_a_dokumentum**-nak (here document) hívjuk; ez azt jelenti, hogy a bemenet itt van és nem valahol egy fájlban. A << jelzi ezt a szerkezetet; az ezt követő szó (példánkban az End) határolja a bemenetet. Bemenet mindaz, ami csak a megadott szóból álló sor előtt található. A burok az **itt_a_dokumentum**-ban behelyettesíti a *$*, `...` és \ jeleket, kivéve, egy része idézőjelek vagy fordított törtvonalak közé van zárva. Az utóbbi esetben az egész dokumentumot szóról szóra veszi.

3.2. táblázat

Be- és kimenetátirányítások a burokban

>fájl	a szabványos kimenetet a *fájl*-ba irányítja
>>fájl	a szabványos kimenetet a *fájl*-hoz hozzáfűzi
<fájl	a szabványos bemenetet a *fájl*-ból veszi
$p_1 \mid p_2$	a p_1 program szabványos kimenetét p_2 bemenetére teszi
^	| szinonimája
n>fájl	az *n* fájlleíró kimenetét a *fájl*-ba irányítja
n>>fájl	az *n* fájlleíró kimenetét a *fájl*-hoz hozzáfűzi
n>&m	az *n* fájlleíró kimenetét az *m* fájlleíró kimenetével összeválogatja
n<&m	az *n* fájlleíró bemenetét az *m* fájlleíró bemenetével összeválogatja
<<s	*itt_a_dokumentum*: veszi a szabványos bemenetet a következő sorkezdő *s*-ig; $, `...` és \ helyettesítést végez
<<\s	*itt_a_dokumentum* helyettesítés nélkül
<<'s'	*itt_a_dokumentum* helyettesítés nélkül

A fejezet végén egy sokkal érdekesebb példa kapcsán még visszatérünk az **itt_a_dokumentum**-hoz.

A 3.2. táblázatban felsoroltuk a különböző ki- és bemenetátirányításokat.

3-14. feladat. Hasonlítsuk össze a **411** program **itt_a_okumentum** változatát az eredetivel! Melyiket könnyebb karbantartani? Melyik jobb alap egy általános szolgáltatáshoz? □

3.8. Burok-programciklusok

A burok valójában egy programozási nyelv; vannak változói, ciklusai, elágazási lehetőségei stb. Most az egyszerű ciklusszervezést mutatjuk be, a vezérlés ellenőrzését részletesebben az 5. fejezetben tárgyaljuk.

Több fájlnév ciklusba szervezése nagyon gyakori. A burok **for** utasítása az egyetlen olyan vezérlőutasítás, amelyet egyszerűen begépelhetünk a terminálon anélkül, hogy későbbi végrehajtása céljára egy fájlba kellene írnunk. A szintaxis a következő:

for *változó* **in** *szavak listája*
do
        parancsok
done

Következő példánk egy olyan **for** utasítás, amely soronként egy fájlnevet visszhangzik:

```
$ for i in *
> do
>     echo $i
> done
```

Az *i* helyén bármely burokváltozó állhat, de *i* a szokásos. Figyeljük meg, hogy a változó értékét a **$i** adja meg, de a **for** a ciklusváltozóra **i** néven hivatkozik! Argumentumként a *-ot használtuk, amivel a munkakatalógus összes fájlját megkapjuk, de bármilyen más argumentumlistát is használhatunk. Normális esetben érdekesebb dolgokat akarunk csinálni, mint egyszerűen fájlneveket kiíratni. Gyakori alkalmazás, hogy egy csomó fájlt összehasonlítunk a korábbi változataikkal. Pl. összehasonlítjuk a 2. fejezet régi változatát (amely az **old** jegyzékben van) a mostanival:

```
$ ls sz2.* | 5
sz2.1 sz2.2 sz2.3 sz2.4 sz2.5
sz2.6 sz2.7
$ for i in sz2.*
> do
> echo $i:
```

```
> diff -b old/$i $i
> echo                                  Egy üres sor a jobb olvashatóság kedvéért.
> donne | pr -h "diff `pwd`/old `pwd`" | lpr &
3712                                    Folyamatazonosító.
$
```

A kimenetet a **pr** és **lpr** programokhoz „csöveztük", csupán azért, hogy ezt a lehetőséget is bemutassuk: a **for**-on belüli programok szabványos kimenete magának a **for**-nak a szabványos kimenetére megy. A **pr** program **-h** opciójával két **pwd** hívást tartalmazó ügyes fejlécet viszünk a kimenetre. Az egész parancssorozatot háttérben (&) futtatjuk, így nem kell várnunk rá; az & a teljes ciklusra és csőhálózatra vonatkozik.

Mi a fenti formában szeretjük a **for** utasítást, de lehet tömörebben is írni, csupán arra kell ügyelni, hogy a **do** és a **done** kulcsszavak csak közvetlenül újsor vagy pontosvessző után (azoktól jobbra) állhatnak. Ha a **for** terjedelme megengedi, jobb az egészet egy sorba írni:

for i in *lista*; **do** *parancsok*; **done**

A **for** ciklust kell használnunk parancsok többszöri elvégeztetésére és akkor, ha az egyes parancsokba beépített argumentumfeldolgozás nem alkalmazható. Ne használjuk viszont akkor, ha a parancs maga is ciklusban végigmegy a fájlneveken:

```
# szerencsétlen gondolat:
for i in $*
do
chmod +x $i
done
```

sokkal rosszabb megoldás, mint a

```
chmod +x $*
```

mert a **for** ciklus minden fájlra egy külön **chmod**-ot hajt végre, ami sokkal több erőforrást köt le. (Győződjünk meg arról, hogy megértettük, mi a különbség a

```
for i in *
```

és a

```
for i in $*
```

között! Az előbbi a munkajegyzék összes fájlnevével, az utóbbi a burokfájl összes argumentumával ismétlődik.)

A **for** argumentumlistáját gyakran minta szerinti fájlnévkeresésből kapjuk, de származhat bárhonnan. Lehet ilyen:

```
$ for i in `cat ...`
```

vagy egyszerűen csak begépelhetjük az argumentumokat. Pl. ebben a fejezetben korábban létrehoztunk több oszlopban nyomtató programokat, amelyeknek a **2, 3,** ... stb. neveket adtuk. Ezek egy fájlhoz tartozó láncszemek, amelyeket – ha a **2** fájlt már megírtuk – létrehozhatunk így:

```
$ for i in 3 4 5 6; do ln 2 $i; done
$
```

A **for** egy érdekesebb alkalmazásaként a **pick** segítségével válasszuk ki azokat a fájlokat, amelyeket össze kell hasonlítani a háttérjegyzékben levőkkel:

```
$ for i in `pick sz2.*`
> do
>       echo $i:
>       diff old/$i $i
> done | pr | lpr
sz2.1? y
sz2.2?
sz2.3?
sz2.4? y
sz2.5? y
sz2.6?
sz2.7?
$
```

Célszerű ezt a ciklust egy burokfájlba tenni, így legközelebb nem kell újra begépelni: amit kétszer kell csinálni, arra harmadszor is szükség lesz!

3-15. feladat. A **diff** ciklust burokfájlba tettük. Tehetjük-e a **pick**-et is burokfájlba? Hogyan? Vagy miért nem? □

3-16. feladat. Mi történik, ha a fenti ciklus utolsó sora

```
> done | pr | lpr &
```

vagyis & jelre végződik? Próbáljuk ki! □

3.9. Kössünk össze mindent: bundle

Hogy ízelítőt adjunk arról, hogyan fejlődnek a burokfájlok, csináljunk végig egy hosszabb példát! Képzeljük el, hogy egy másik gépen dolgozó ismerősünktől, mondjuk **valahol!jani**-tól* levelet kaptunk. Ebben másolatot kér tőlünk a **bin** jegyzékünkben levő burokfájljainkról. A legegyszerűbb, ha a listát postafordultával levélben átküldjük, úgyhogy már kezdhetjük is a gépelést:

* Számtalan jelölés van a távoli gép címeinek megadására. A legelterjedtebb a gép!személy formula. L. **mail(1).**

```
$ cd /usr/you/bin
$ for i in `pick *`
> do
>         echo = = = = = Ez a $i fájl = = =
>         cat $i
> done | mail valahol!jani
$
```

Nézzük meg a dolgot **valahol!jani** szempontjából: ő kapni fog egy levelet, amelyben világosan elválasztva benne lesz az összes fájl, de ezután egy szerkesztővel szét kell tördelni azt az összetevő fájljaira. Jobb megoldás volna, ha a jól megalkotott levél automatikusan kicsomagolná magát, és a címzettnek nem volna vele dolga. Ebből következik, hogy a levélnek egy burokfájlnak kell lennie, ami a fájlokat és a kicsomagolási utasítást is tartalmazza.

A másik megoldás, hogy a burok **itt_a_dokumentum**-jai kényelmesen összekapcsolják a parancs hívását és a parancshoz tartozó adatokat. Munkánk csupán annyi, hogy az idézőjeleket jól helyezzük el. Íme egy **bundle** nevű működő program, amely a szabvány kimenetén összeszedi a fájlokat egy önmagát magyarázó burokfájlban.

```
$ cat bundle
# bundle: a fájlokat összecsomagoljuk
echo '# szétszedéshez burkot a fájlnak'
for i
do
    echo "echo $i 1>&2"
    echo "cat >$i <<'$i vége'"
    cat $i
    echo "$i vége"
done
$
```

A $i vége idézőjelbe tétele biztosítja, hogy a fájlokban levő összes burok-metakarakter értelmezetlenül maradjon.

Természetesen okosan tesszük, ha kipróbáljuk, mielőtt **valahol!jani**-t megkínozzuk vele:

```
$ bundle cx lc >vacak                    Csináljunk "próbabatyut"!
$ cat vacak
# szétszedéshez burkot a fájlnak
echo cx 1>&2
cat >cx <<'cx vége'
chmod +x $*
cx vége
echo lc 1>&2
cat >lc <<'lc vége'
# lc: a fájl sorainak száma
vc -l $*
lc vége
$ mkdir test
```

```
$ cd test
$ sh ../vacak                          Próbáljuk ki!
cx
lc
$ ls
cx
lc
$ cat cx
chmod +x $*
$ cat lc
# lc: a fájl sorainak száma
wc -l $*                               Jónak látszik.
$ cd ..
$ rm vacak test/*; rmdir test          Törlés.
$ pwd
/usr/you/bin
$ bundle `pick *` | mail valahol!jani  Elküldjük a fájlokat.
```

Csak akkor van baj, ha olyan fájlt küldünk, amely történetesen tartalmaz egy

```
fájlnév vége
```

alakú sort, ez azonban igazán nem nagyon valószínű. Ahhoz, hogy a **bundle** teljesen biztonságos legyen, még néhány dolgot meg kell majd ismernünk, de már ebben a formában is jól használható és kényelmes.

A **bundle** jól mutatja a UNIX környezet rugalmasságát: használja a burokciklusokat, a bemenet/kimenet átirányítását, az **itt_a_dokumentum**-okat, a burokfájlokat, közvetlenül kapcsolódik a **mail**-hez, és ami talán a legérdekesebb, olyan program, amely saját maga létrehoz programokat. Eddig a **bundle** a legformásabb burokprogram, ez a néhány sornyi kód valamit nagyon egyszerűen, hasznosan és elegánsan csinál.

3-17. feladat. Hogyan küldhetjük el a **bundle** segítségével egy jegyzék és annak aljegyzék összes fájlját? Útmutatás: a burokfájlok lehetnek rekurzívak. □

3-18. feladat. Módosítsuk a **bundle**-t úgy, hogy minden fájllal együtt tartalmazza az **ls −l** által szolgáltatott információt, különösen az engedélyeket és az utolsó változtatás idejét! Hasonlítsuk össze a **bundle** és az **ar(1)** archív programok lehetőségét! □

3.10. Miért programozható burok?

A UNIX burok nem egy tipikus parancsértelmező: a parancsok szokásos futtatásánál sokkal többet tudó programnyelv. Érdemes kicsit visszapillantani arra a sok mindenre, amit láttunk, hiszen nagyon sok anyag volt ebben a fejezetben. Ráadásul azt

ígértük, hogy „általánosan használt, ügyes lehetőségekről" fogunk beszélni, és ezután oldalakon keresztül soroltuk a burokprogramozási példákat. Amikor a burkot használjuk, állandóan kis egysoros programokat írunk: a csőhálózat is egy program, a „kész a tea" példánk is program. A burok állandó programozása azonban (ha egyszer más megismerkedtünk vele) olyan egyszerű és természetes, hogy már nem is gondolunk programozásra.

A burok elvégzi a ciklusszervezést, a be- és kimenet átirányítását a < és > jelekkel és a fájlnév kiterjesztését a *-gal. A programoknak ezekkel nem kell foglalkozniuk, és ami még fontosabb: ezeket a lehetőségeket minden program egyformán használja. További lehetőségeket, mint pl. a burokfájlok és a csövek lehetőségét a mag nyújtja, a burok csupán természetes szintaxist ad a létrehozásukhoz. Ezek a lehetőségek nem csupán a kényelmet szolgálják, valóban növelik a rendszer használhatóságát.

A burok ereje és kényelme nagyrészt az alatta levő UNIX magból ered; pl. bár a csöveket a burok hozza létre, de a mag viszi át az adatokat rajta. Az a módszer, ahogy a rendszer a végrehajtható fájlokat működteti, lehetővé teszi, hogy burokfájlokat írjunk. Ezek pontosan ugyanúgy futnak, mint a lefordított programok. A felhasználónak nem is kell tudnia, hogy ezek parancsfájlok, amelyeket nem **RUN** jellegű speciális paranccsal kell hívni. Továbbá a burok maga is egy program, de nem része a magnak, ezért alakítható, kiterjeszthető, ugyanúgy használható, mint bármely más program. Ez a gondolat a UNIX rendszerben sem teljesen egységes, de sokkal erősebb, mint bármely más rendszerben.

Az 5. fejezetben még visszatérünk a burokprogramozás témájához, de mindig gondoljunk arra, hogy bármit csinálunk a burokkal, mindig programozzuk azt – azért működik ilyen jól.

Történeti és irodalmi megjegyzések

A burok a UNIX rendszerben kezdettől fogva programozható. Eredetileg külön **if, goto** parancsok és címkék voltak; a **goto** parancs úgy működött, hogy végigfutott a bemeneti fájlon és megkereste a megfelelő címkét. (Miután egy csövet nem lehet újraolvasni, lehetetlen volt olyan burokfájlhoz hozzákapcsolódni, amelyben vezérlésátadás is volt.)

A 7. kiadás burokprogramját eredetileg Steve Bourne írta, felhasználva John Mashey néhány ötletét és segítségét. Ebben minden benne volt, ami a programozáshoz kell (ezt az 5. fejezetben látni fogjuk). A továbbiakban ésszerűsítették a bemenetet és kimenetet: lehetővé tették a burok programokba, ill. programokból való minden korlátozástól mentes átirányítást. A fájlnév-metakarakterek elemzése is a burok belső feladata. A korábbi változatokban ez önálló program volt, ami nagyon kis gépeken is megélhetett.

Egy másik fontos burok, amivel találkozhatunk, a **csh,** az ún. *C-burok,* amelyet Berkeley-ben a 6. kiadás burkára építve Bill Joy fejlesztett ki (lehetséges, hogy Olvasóink éppen ezt használják). A C-burok továbblépett az interaktivitás irányában, mint a Bourne-burok – a legjelentősebb benne egy emlékeztető mechanizmus, amely (némi szerkesztés árán) lehetővé teszi a korábban használt parancsok rövidített

ismétlését. A két burok szintaxisa egy kicsit eltér. Mivel a C-burok egy korábbi burkon alapszik, kevésbé kényelmes a programozása, inkább interaktív parancsértelmező, mint programnyelv. Különös hiányossága, hogy a vezérlést ellenőrző szerkezetekhez csövekkel nem lehet kapcsolódni.

A **pick** Tom Duff találmánya, a **bundle**-t egymástól függetlenül Alan Hewett és James Gosling alkották meg.

4. Szűrők

Sok olyan UNIX program van, amely valamilyen bemenetről olvas, végrehajt valami egyszerű átalakítást, majd valamilyen kimenetre ír. Ilyenek pl. a **grep** vagy a **tail,** amelyek a bemenet bizonyos részét kiválasztják, a **sort,** amely rendezi, a **wc,** amely megszámlálja a bemenetet stb. Az ilyen programokat *szűrőknek* hívjuk.

Ebben a fejezetben a leggyakrabban használt szűrőket tárgyaljuk. A **grep**-pel kezdjük, de az 1. fejezetben látottaknál sokkal bonyolultabb mintákkal. Bemutatjuk a **grep** család két további tagját: az **egrep** és az **fgrep** programokat.

A következő szakaszban néhány további hasznos szűrőt tárgyalunk, köztük a karaktereket konvertáló **tr,** a más rendszerekből származó adatok kezelésére szolgáló **dd** és az ismétlődő szövegsorokat kereső **uniq** programokat. A **sort** programmal is részletesebben foglalkozunk, mint azt az 1. fejezetben tettük.

A fejezet hátralevő részét két általánosan használható „adatátalakítónak" vagy „programozható szűrőnek" szenteljük. Azért mondjuk programozhatónak őket, mert a végrehajtandó transzformációt egy egyszerű programnyelven, program formájában kell megadni. Különböző programokkal nagyon különböző transzformációkat lehet megvalósítani.

Az egyik program a **sed,** amely a *stream editor* szavakból, a másik az **awk,** amely a szerzőiről kapta a nevét. Mindkettő a **grep** általánosításával jött létre:

```
$ program mintatevékenység fájlnevek ...
```

A **grep** sorban végignézi a fájlokat és minta szerint egyező sorokat keres. Ha ilyet talál, akkor végrehajtja a megfelelő tevékenységet. A **grep** számára a minta szabályos kifejezés (ugyanúgy, mint az **ed**-nél), a tevékenység a mintával egyező sorok kiírása.

A **sed** és az **awk** a mintát is és a tevékenységet is általánosítja. A **sed** az **ed** leszármazottja, vesz egy szerkesztőparancsokból összeállított „programot", és a fájlokból jövő adatáradat minden sorára végrehajtja a program parancsait. Az **awk** kevésbé alkalmas szöveg átalakítására, mint a **sed,** viszont vannak benne aritmetikai műveletek, változók, beépített függvények, és egy C jellegű programnyelv. Ebben a fejezetben egyik programról sem adunk teljes leírást, az megtalálható a kézikönyv 2B kötetében.

4.1. A grep család

Az 1. fejezetben röviden már említettük a **grep**-et, és azóta többször használtuk is:

$ grep minta fájlnevek...

Ez a parancssor a megnevezett fájlokat vagy a szabványos bemenetet vizsgálva kiírja azokat a sorokat, amelyekben a minta megjelenik. A **grep** sokat segít, ha programokban változókat vagy szövegben szavakat keresünk, vagy ha egy program kimenetéből részeket akarunk kiválasztani:

*$ grep −n variable *.[sz]*	Megkeresi a **variable** előfordulásait a C forrásban.
$ grep From $MAIL	Kiírja a postaládából a levélfejléceket.
$ grep From $MAIL / grep −v mary	Kiírja a nem **mary**-től érkezett levélfejléceket.
$ grep −y mary $HOME/lib/telefonkönyv	Megkeresi **mary** telefonszámát.
$ who / grep mary	Megnézi, hogy **mary** bejelentkezett-e?
$ ls / grep −v múlandó	Kiírja a **múlandó** -t nem tartalmazó fájlneveket.

A **−n** opció kiírja a sorszámokat, a **−v** nemegyezést keres, a **−y** a minta kisbetűit a fájl kis- és nagybetűivel egyezteti (a nagybetűket csak a nagybetűkhöz hasonlítja).

Az összes eddigi példánkban a **grep** mintái betűkből és számokból álló egyszerű karakterláncok voltak. A **grep** ennél sokkal bonyolultabb formában megadott mintákat is tud keresni: értelmezni tud egy egyszerű karakterlánc-leíró nyelven írt kifejezéseket.

A minták gyakorlatilag kötetlen formájú karakteres kifejezések, amelyeket *szabályos kifejezéseknek* (regular expressions) nevezünk. A **grep** ugyanazokat a szokásos kifejezéseket érti, mint az **ed**; eredetileg ugyanis a **grep** (egyetlen este műveként) az **ed**-ből származik.

A szabályos kifejezéseket úgy adjuk meg, hogy − a burok által használt * stb. jelekhez hasonlóan − különleges jelentést adunk bizonyos karakternek. Van néhány új metakarakter, és sajnos néhány régi, megváltozott jelentéssel. A 4.1. táblázatban felsoroljuk a szabályos kifejezések metakaraktereit, de előbb röviden áttekintjük őket.

A ^ és *$* metakarakterek a mintát a sor elejére (^) vagy a végére (*$*) rögzítik. Pl. a

$ grep From $MAIL

parancssor a postánkban megkeresi azokat a sorokat, amelyek tartalma From, viszont a

$ grep '^From' $MAIL

parancssor azokat a sorokat írja ki, amelyek kezdete From (nagyon valószínű, hogy ezek üzenet-fejlécsorok). A szabályos kifejezések metakarakterei között vannak olyanok, amelyek a buroknak is metakarakterei, ezért általában célszerű a **grep** mintákat fél idézőjelek közé tenni.

A **grep** a burokhoz hasonlóan *karakterosztályokat* is elfogad, pl. az *[a−z]* mintának minden kisbetű megfelel. Megadásukban azonban vannak különbségek: ha egy **grep** karakterosztálya kalappal (^) kezdődik, akkor a mintának az osztályba tartozók *kivételével* minden karakter megfelel. Így a *[^0−9]* mintának minden nemnumerikus karakter megfelel. Továbbá a burok karakterosztályában a fordított törtvonal védi a] és − jeleket, az **ed** és a **grep** viszont megköveteli, hogy e karakterek egyértelmű jelentésűek legyenek. Pl. a *[][−]* mintának megfelel a szögletes nyitó vagy záró zárójel és a mínuszjel.

A pont (.) jelentése azonos a burok kérdőjelével (*?*): minden karakterrel egyezést mutat. (Valószínűleg a pont az a karakter, amelynek a legkülönbözőbb jelentései vannak a különböző UNIX programokban.) Az elmondottakat a következő két példával szemléltetjük:

$ *ls −l | grep '^d'*	Kiírja az aljegyzékek neveit.
$ *ls −l | grep '^.....rw'*	Kiírja azokat a fájlokat, amelyeket a többiek írhatnak és olvashatnak.

A kalap (^) jel és a hét pont a sor elején bármely hét karakterrel megegyezik, ami az **ls −l** kimenetén tetszőleges engedély karakterláncot jelent.

A csillag (*) végjel (closure operator) egy szabályos kifejezés bármely karaktere vagy metakaraktere után (lehet karakterosztály is) állhat. Hatására a **grep** tetszőleges számú egymást követő, a végjel előtt kiírt vagy metakarakterrel adott karakter egyezése esetén mutat egyezést. Pl. a x* tetszőleges hosszúságú x sorozatnál; az *[a−zA−Z]** bármely alfabetikus karakterláncnál; a .* az újsorig bárminél; és a .** bármit tartalmazó, x-re végződő sornál mutat egyezést.

A * végjellel kapcsolatban néhány fontos dolgot meg kell jegyeznünk. Először: a * csak egy karakterre vonatkozik, így az *xy** olyan mintát keres, amelyben egy x-et y karakterek sorozata követ, nem pedig xyxyxy jellegű sorozatot. Másodszor: a tetszőleges számú karakter nulla is lehet, ezért ha legalább egy karaktert akarunk a mintával egyeztetni, akkor meg kell dupláznunk azt. Például egy betűsorozat egyeztetéséhez a helyes kifejezés: *[a−zA−Z] [a−zA−Z]** (egy betű, amelyet nulla számú vagy annál több betű követ). A burok * fájlnév-azonosító karaktere megegyezik a .* szabályos kifejezéssel.

A **grep**-nek nincs újsor-karaktert egyeztető szabályos kifejezése, az egyezést soronként vizsgálja.

A szabályos kifejezésekkel a **grep** egy egyszerű programnyelv. Például emlékezzünk vissza, a jelszófájl második mezője a titkosított jelszó! A következő parancs megkeresi a jelszó nélküli felhasználókat:

$ *grep '^[^:]*::' /etc/passwd*

A megadott minta: soreleje, tetszőleges számú karakter, amely nem kettőspont, dupla kettőspont.

A **grep** a legrégibb programcsalád, tagjai az **fgrep** és az **egrep.** Ez utóbbiak alapvetően ugyanúgy viselkednek, mint a **grep,** de az **fgrep** több karakterláncot párhuzamosan keres, az **egrep** pedig a **grep**-hez hasonlóan igazi szabályos kifejezése-

A **grep** és **egrep** szabályos kifejezései
(csökkenő precedenciarendben)

c	bármely nem speciális c karakter saját magával egyezik
\c	a c karakter speciális jelentését kikapcsolja
^	sor eleje
$	sor vége
.	bármely egyetlen karakter
[...]	bármely ... közüli egyetlen karakter; a–z tartomány megengedett forma
[^...]	bármilyen egyetlen karakter, amely nincs a ... között
\n	ami az n-edik \ karakter (...\); tartományok megengedettek (csak **grep**)
r*	r nulla vagy több előfordulása
r+	r egy vagy több előfordulása (csak **egrep**)
r?	r nulla vagy egy előfordulása (csak **egrep**)
r1r2	r1-et követi r2
r1|r2	r1 vagy r2 (csak **egrep**)
\(r\)	r jelzett szabályos kifejezés (csak **grep**), egymásba ágyazható
(r)	r szabályos kifejezés (csak **egrep**); egymásba ágyazható
	Az újsorral semmilyen szabályos kifejezés sem egyezik

ket értelmez, de van VAGY művelet, és a kifejezések – ahogy azt látni fogjuk – csoportosíthatók.

A **–f** opcióval kiegészített **fgrep** és **egrep** parancs fájlból olvassa a mintát. Ebben a fájlban a párhuzamosan keresendő mintákat újsor-karakterek választják el. Ha vannak olyan szavak, amelyeket rendszeresen hibásan írunk, akkor a szövegeinket az **fgrep** parancs **–f** opciójával kényelmesen ellenőriztethetjük úgy, hogy ezeket a szavakat egy fájlban, soronként egyesével elhelyezzük:

```
$ fgrep –f szokásoshibák szövegfájl
```

Az **fgrep** ugyanazokat a szabályos kifejezéseket értelmezi, mint a **grep,** de van néhány további lehetősége is (l. a 4.1. táblázatot): a kerek zárójelek csoportosítást jelentenek, így az *(xy)** minta megegyezik minden üres karakterlánccal és az xy, xyxy, xyxyxy

stb. karakterláncokkal; a két függőleges vonal (*|*) VAGY műveletet jelent, így a *tegnap|holnap* mintának megfelel a tegnap vagy a holnap (ugyanez a minta megadható a *(teg|hol)nap* alakban is). Végül, az **egrep**-ben van két további végjel, a + és a *?*. Az *x+* mintának egy vagy több x, az *x?* mintának nulla vagy legfeljebb egy x felel meg.

Az **egrep** kiválóan használható olyan szójátékoknál, amikor a szótárban adott tulajdonságú szavakat kell keresnünk. A mi gépünkön megvan a Webster's Second International szótár, definíciók nélkül, soronként egy szót tartalmazó formában. Olvasóink gépén is lehet valamilyen szótár, mondjuk a helyesírás ellenőrzésére: **/usr/dict/words** néven. A következő mintával azokat a szavakat kereshetjük meg, amelyekben ábécésorrendben mind az öt magánhangzó előfordul.

```
$ cat magánhangzók
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$
$ egrep -f magánhangzók /usr/dict/web2 |3
abstemious          abstemiously        abstentious
acheilous           acheirous           acleistous
affectious          annelidous          arsenious
arterious           bacterious          caesious
facetious           facetiously         fracedinous
majestious
$
```

A **magánhangzók** fájlban a minta nincs idézőjelek között. Ha az **egrep** mintákat idézőjelek közé tesszük, akkor a burok megakadályozza a parancsok értelmezését, de leszedi az idézőjeleket, így az **egrep** sosem látja meg őket. A burok azonban nem vizsgálja a fájlt, ezért annak tartalmát *nem* kell idézőjelek közé tenni. Ebben a példában a **grep**-et is használhattuk volna, de végjeleket tartalmazó mintáknál az **egrep** sokkal gyorsabb, különösen, ha nagy fájlokat vizsgálunk.

Következő példánkban megkerestetjük az összes olyan legalább hatbetűs szót, amelyben a betűk ábécésorrendben követik egymást:

```
$ cat monoton
^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$
$ egrep -f monoton /usr/dict/web2 | grep '.....' | 5
abdest      acknow      adipsy      agnosy      almost
befist      behint      beknow      bijoux      biopsy
chintz      dehors      dehort      deinos      dimpsy
egilops     ghosty
$
```

Figyeljük meg, hogyan szűrjük a **grep**-pel az egrep kimenetét!

Miért van három **grep** program? Az **fgrep** nem értelmez metakaraktereket, de ezernyi szót tud hatékonyan párhuzamosan keresni (ha már elindult, futási ideje független a keresendő szavak számától), ezért elsősorban könyvtári kereséshez hasonló feladatokra használják. Az **fgrep**-pel lényegesen nagyobb mintákat lehet keresni, mint amekkorák az **egrep** és **grep** algoritmusaival kezelhetők. A **grep** és **egrep** közötti

különbséget nehezebb bemutatni. A **grep** sokkal korábban keletkezett, mint az **egrep**, az **ed** szabályos kifejezéseihez hasonlókat használ, és vannak jelzett (tagged) szabályos kifejezései. A **grep** opciókészlete bővebb. Az **egrep** sokkal általánosabb kifejezéseket értelmez (kivéve a jelzett kifejezéseket), és jelentősen gyorsabban (a mintától független sebességgel) fut mint a **grep**, de a szabványos változat indítása bonyolult mintánál hosszadalmas. Egy újabb **egrep** változat már azonnal indul, úgyhogy a **grep** és az **egrep** egyetlen mintakereső programként összekapcsolódik.

4-1. feladat. Keressük meg a jelzett szabályos kifejezéseket az 1. függelékben vagy az ed(1)-ben, és keressünk a **grep**-pel polindromokat – olyan szavakat, amelyek mindkét irányból olvasva azonos szót adnak! Útmutatás: minden szóhosszúságra külön mintát írjunk! □

4-2. feladat. A **grep** úgy működik, hogy a bemenetről olvas egy sort, abban keresi a mintával való egyezést, majd veszi a következő sort. Hogyan működne a **grep**, ha a szabályos kifejezések újsor-karaktert is tartalmazhatnának? □

4.2. További szűrők

Ebben a szakaszban a rendszerben levő kisebb szűrők gazdag választékával és azok lehetőségeivel ismerkedünk meg, valamint alkalmazásukra láthatunk néhány példát. A lista nem teljes – már a 7. kiadásban is több van, ráadásul minden változat hozott létre saját szűrőket. A szabványos szűrők leírása megtalálható a kézikönyv 1. részében.

Kezdjük az ismerkedést a **sort** paranccsal, amely valószínűleg a leghasznosabb szűrő! Alapjait már az 1. fejezetben elmondtuk: bemenetét soronként ASCII kódok szerint rendezi. Nyilvánvalóan ez az alapértelmezés, de számos más módon is lehet az adatokat rendezni. Erre szolgálnak a **sort** különböző opciói. Például: a **−f** opció a kis- és nagybetűk megkülönböztetése nélkül rendez; a **−d** opció (szótári rend) a rendezéskor csak a betűket, számokat és szóközöket veszi figyelembe.

Bár az ábécé szerinti sorrend a leggyakoribb, néha a számjegyek szerinti sorrend is előfordul. A **−n** opcióval a **sort** számérték szerinti sorrendbe rendez, a **−r** opció minden összehasonlítás értelmét az ellenkezőjére fordítja. Így pl.:

```
$ ls | sort -f        A fájlneveket ábécé szerint rendezi.
$ ls -s | sort -n     A legkisebb fájllal kezdi a rendezést.
$ ls -s | sort -nr    A legnagyobb fájllal kezdi a rendezést.
```

A **sort** parancs általában a bemenet teljes sorát rendezi, de meg lehet mondani neki azt is, hogy csak adott mezők szerint rendezzen. A *+m* a parancs számára azt jelenti, hogy az összehasonlításból maradjon ki az első m mező; a *+0* a sor elejét jelenti. Például:

```
$ ls -l | sort +3nr    Bájtszám szerint rendez, a legnagyobbal kezd.
$ who | sort +4n       Bejelentkezési idő szerint rendez, a legrégibbel kezd.
```

A **sort** parancs jól használható opciója még a **−o,** amely a kimenet számára megad egy fájlnevet (ez lehet az egyik bemeneti fájl is), és a **−u,** amely a rendezési mezőikben megegyező sorok közül csak egyet hagy meg.

Egyszerre több rendezési kulcsot is megadhatunk. Ezt mutatja a kézikönyv sort (1) lapjának itt következő ravasz példája:

```
$ sort +0f +0 −u fájlnevek
```

A *+0f* kulccsal a **sort** úgy rendezi a sorokat, hogy a rendezés során nem tesz különbséget kis- és nagybetű között, így az azonos sorok nem feltétlenül lesznek szomszédosak. Ezért a második (*+0*) kulcs az első rendezésből származó azonos sorokat normál ASCII sorrendbe rakja. Végül a **−u** elhagy minden szomszédos duplikátumot. Tehát ez a parancs egy szólistát (minden sorban egy szó) szóismétlés nélkül, egyértelműen kinyomtat. E könyv eredeti tárgymutatója hasonló, a **sort** még több lehetőségét kihasználó rendezőparanccsal készült. L. sort(1).

A **sort** parancs **−u** opcióját a **uniq** parancs sugallta, amely az egymás utáni egyforma sorok csoportjából csak egyet hagy meg. Az önálló **uniq** programmal ez a funkció a rendezéstől függetlenül is elvégeztethető. A **uniq** használható pl. a többszörös üres sorok törlésére akár rendezett a bemenete, akár nem. Opcióival a duplikátumokat többféleképpen kezelhetjük: **uniq −d** csak a duplikált sorokat írja ki; **uniq −u** csak az egyértelmű (nem duplikált) sorokat írja ki; **uniq −c** megszámlálja, hogy az egyes sorok hányszor fordulnak elő. (Rövidesen látunk erre példát.)

A **comm** parancs két rendezett fájlt összehasonlító program. Ha adott az **f1** és **f2,** két rendezett bemeneti fájl, akkor a **comm** háromoszlopos kimenetet ad: első oszlopban a csak az elsőben meglevő sorokat; második oszlopban a csak a másodikban meglevő sorokat; harmadik oszlopban a mindkét fájlban meglevő sorokat. Egy opcióval bármelyik oszlopot elhagyhatjuk:

```
$ comm −12 f1 f2
```

csak a mindkét fájlban meglevő sorokat írja ki,

```
$ comm −23 f1 f2
```

csak azokat a sorokat írja ki, amelyek az elsőben megvannak, de a másodikban nem. Ez hasznos jegyzékek összehasonlításánál vagy egy szólista és egy szótár összevetésénél.

A **tr** parancs átváltoztatja a bemenet karaktereit. Leghétköznapibb alkalmazása a kisbetű−nagybetű átírás:

```
$ tr a−z A−Z    A kisbetűket nagybetűkre írja át.
$ tr A−Z a−z    A nagybetűket kisbetűkre írja át.
```

A **dd** parancs alapvetően eltér az eddig megismert parancsoktól. Eredetileg más rendszerből érkező szalagos adatok feldolgozására szánták − maga a név az OS/360 munkavezérlő (job control) nyelvéből származik. A **dd** paranccsal végezhetünk kisbe-

tű – nagybetű átírást (egészen más szintaxissal, mint a **tr**), ASCII és EBCDIC kódok közötti átalakítást, nem UNIX jellegű, szóközökkel feltöltött, rögzített méretű adatrekordok olvasását vagy írását. A **dd**-nek van egy sor bináris adatkezelő lehetősége is, ezért gyakran használjuk nyers, formátlan adathalmaz kezelésére.

Most bemutatjuk, mi mindent lehet megvalósítani szűrők kombinálásával. A következő csőhálózat kiírja a bemenet 10 leggyakoribb szavát:

```
cat $* |
tr -sc A-Za-z '\012' |      Betűtől különböző karakterek helyett újsor.
sort |
uniq -c |
sort -n |
tail |
5
```

A **cat** összegyűjti a fájlokat, mivel **tr** csak a szabvány bemenetet olvassa. A kézikönyvből vett **tr** parancs a szomszédos nem betűk karaktereket átírja újsor-karakterré, ezáltal a bemenet minden sorában egy szó lesz. Ezután a szavakat rendezzük, majd a **uniq -c** az azonos szavakat egy sorban összesíti eléírva, hogy hányszor fordult elő a szó. Ez a szám lesz a **sort -n** parancs rendezési mezője. (A **uniq** előtt és után álló két **sort** olyan gyakran fordul elő, hogy akár idiómának is nevezhetjük.) Eredményként a fájlokban előforduló különböző szavakat előfordulásuk gyakorisága szerint növekvő sorrendbe rendezve kapjuk. A **tail** kiválasztja a 10 leggyakoribb szót (a rendezett lista végét) és az **5** parancs öt oszlopban kiírja azokat.

Figyeljük meg, hogy a | jelre végződő sor folytatható!

4-3. feladat. Az ebben a fejezetben megismert eszközökkel készítsünk egy egyszerű helyesírás-ellenőrző programot a **/usr/dict/words** felhasználásával!

Mik a hiányosságai és azon hogyan lehet segíteni? □

4-4. feladat. Írjunk egy szószámláló programot a legkedvesebb programnyelvünkön, és hasonlítsuk össze annak méretét, sebességét és használhatóságát a szószámláló csőhálózattal! Hogyan lehet azt egyszerűen helyesírás-ellenőrzővé átalakítani? □

4.3. A sed áradatszerkesztő

Foglalkozzunk most a **sed** paranccsal! Mivel ez közvetlenül az **ed** programból származik, könnyen megtanulható és mellesleg **ed** ismereteinket is megerősíti.

A **sed** alapötlete egyszerű:

```
$ sed 'ed parancsok listája' fájlnevek...
```

A bemeneti fájl sorait egyenként beolvassa, alkalmazza rájuk sorra a parancsokat,

és az átszerkesztett formát kiírja a szabványos kimenetre. Például egy fájlhalmazban minden UNIX-ot UNIX(TM)-re cserélhetünk.

```
$ sed 's/UNIX/UNIX(TM)/g' fájlnevek... >output
```

Vegyük észre: a **sed** a bemeneti fájlok tartalmát *nem* változtatja meg. A szabványos kimenetre ír, tehát az eredeti fájlok nem változnak. Most már ismerjük a burkot annyira, hogy megértsük: a

```
$ sed '...' file >file
```

rossz ötlet. A fájl tartalmának kicseréléséhez egy ideiglenes fájlt vagy egy másik programot kell használni. (A későbbiekben lesz szó egy másik programról, amely megvalósítja a létező fájl átírását; nézzük meg az **overwrite** parancsot az 5. fejezetben.)

A **sed** automatikusan kiírja a sorokat, tehát az **s** parancs után nem kell **p**. Ha volna, akkor minden módosított sort kétszer írna ki. Az idézőjel majdnem mindig kell, mert a **sed** metakarakterek nagy része a burok számára is jelent valamit. Például vizsgáljuk meg a fájlneveket generáló **du −a** parancsot! Normálisan a **du** kiírja a méretet és a fájlnevet:

```
$ du −a sz4.*
18      sz4.1
13      sz4.2
14      sz4.3
17      sz4.4
2       sz4.9
$
```

sed paranccsal a méretet a kiírásból elhagyhatjuk, de a szerkesztőparancsot idézőjelbe kell tenni, nehogy a *-ot és a tabulátort a burok értelmezze:

```
$ du −a sz4.* | sed 's/.*→//'
sz4.1
sz4.2
sz4.3
sz4.4
sz4.9
$
```

Az **s** parancs törli az összes karaktert (.*) a jobb szélső tabulátorig, beleértve a tabulátort is (a mintában ezt → jelzi).

Hasonlóan válogathatjuk a felhasználók neveit és a bejelentkezésük idejét a **who** kimenetéből:

```
$ who
lr      tty1   Sep 29 07 : 14
ron     tty3   Sep 29 10 : 31
you     tty4   Sep 29 08 : 36
td      tty5   Sep 29 08 : 47
```

```
$ who | sed 's/ .* / /'
lr 07 : 14
ron 10 : 31
you 08 : 36
td 08 : 47
$
```

Az **s** parancs a szóközt és mindent, ami a következő szóközig utána áll (a többszörös szóközöket is beleértve), egyetlen szóközre cseréli. Itt is kell az idézőjel.

Majdnem ugyanilyen **sed** parancsot használhatunk a **getname** program alkalmazásakor, amely kiírja saját felhasználói nevünket:

```
$ cat getname
who am i | sed 's/ .*//'
$ getname
you
$
```

Még egy **sed** parancs, amelyet olyan gyakran használunk, hogy **ind** néven burokfájlt csináltunk belőle. Az **ind** parancs a bemenet elé egy tabulátorstopot tesz. Ez különösen akkor hasznos, ha a jobb elhelyezés miatt valamit odébb akarunk tenni a sornyomtató kiírásában.

Az **ind** megvalósítása egyszerű – egy tabulátort elhelyez minden sor elején:

```
sed 's/^/→/' $*          ind  első változata.
```

Ez a változat az üres sorok elejére is tabulátort tesz, ami feleslegesnek tűnik. A következő, jobb változat kihasználja a **sed** azon képességét, hogy ki tudja válogatni a módosítandó sorokat. Ha a parancs elé egy mintát teszünk, akkor az csak azokra a sorokra lesz hatásos, amelyek megfelelnek a mintának:

```
sed '/./s/^/→/' *$       ind  második változata.
```

A /./ mintával megegyeznek azok a sorok, amelyben az újsor-karakteren kívül legalább egy karakter van, ezekre az **s** parancs végrehajtódik, az üres sorokra nem. Emlékezzünk rá, hogy a **sed** minden sort kivisz, függetlenül attól, hogy változtatásra került-e, így az üres sorok is úgy jelennek meg, ahogy kell!

Még egy **ind**-megírási lehetőség: meg lehet adni, hogy a parancsok csak azokon a sorokon hajtódjanak végre, amelyek nem mutatnak egyezést a mintával. Ehhez a parancs elé egy felkiáltójelet (*!*) kell tenni:

```
sed '/^$/!s/^/→/' $*     ind  harmadik változata.
```

A / *^$*/ mintának az üres sorok felelnek meg (a sor eleje után rögtön a sor vége következik), ezért a / *^$*/*!* jelentése: „az üres sorokon ne hajtsd végre a parancsot"!

A **sed** tulajdonságai között már említettük: minden sort kiír, függetlenül attól,

hogy mi történt vele (kivéve, ha törlődött); az **ed** parancsok nagy része használható. Így könnyű egy olyan **sed** programot írni, amely mondjuk a bemenet első három sorát kiírja, azután kilép:

```
sed 3q
```

Bár a *3q* nem megengedett **ed** parancs, a **sed** számára van értelme: másoljon sorokat, a harmadik után lépjen ki.

További adatátalakításokat is végrehajthatunk, pl. megadhatunk a szövegben bekezdéseket. Ezt megtehetjük úgy, hogy a **sed** kimenetét átfuttatjuk az **ind**-en, de a **sed** többszörös parancsokat is elfogad, ezért megtehetjük a **sed** egyszeri (elég valószínűtlen) behívásával is:

```
sed 's/^/→/
    3q'
```

Figyeljük meg, hol vannak az idézőjelek és az újsor: a parancsoknak külön sorban kell lenniük, de a **sed** a kezdő szóközöket és a tabulátorokat elhagyja.

Az elmondottak után hasznosnak tűnhet egy olyan **head** nevű programot írni, amely minden fájlnévargumentum első néhány sorát kiírja. A **sed 3q** (vagy **10q**) parancsot azonban olyan könnyű beírni, hogy soha nem éreztük a **head** parancs hiányát. Használtuk viszont **ind** programunkat, mert az azzal ekvivalens **sed** parancsot nehezebb beírni. (A könyv írása közben cseréltük fel meglevő 30-soros C nyelvű programunkat az **ind** fenti 2. változatával.) Nem lehet egyértelműen eldönteni, hogy mikor érdemes egy bonyolult parancssor helyett külön parancsot írni; a legjobb szabály, amit találtunk, hogy betesszük saját **bin** jegyzékünkbe, és időnként megnézzük, hogy valóban használtuk-e.

A **sed** parancsokat beírhatjuk egy fájlba és onnan is futtathatjuk:

```
$ sed -f parancsfájl...
```

A kiválasztott sort nem csak sorszámmal (példánkban a 3-mal) adhatjuk meg. Például a

```
$ sed '/minta/q'
```

parancssor az első, mintával egyező sorig, azt is beleértve írja a kimenetet, és a

```
$ sed '/minta/d'
```

parancssor minden olyan sort töröl, amely tartalmazza a mintá-t. A program a kiírás előtt töröl, így a törölt sorok eltűnnek.

A kiírás általában hasznos, de néha felesleges lehet. A **-n** opcióval kikapcsolhatjuk, ez esetben a kimeneten csak a **p** paranccsal kiírt sorok jelennek meg. Például:

```
$ sed -n '/minta/p'
```

ugyanazt csinálja, mint a **grep**. A minta szerinti egyezési feltételt az utána álló *!* jellel megfordíthatjuk, így

```
$ sed -n '/minta/!p'
```

ugyanaz, mint **grep -v** (*sed '/minta/d'* is ugyanez).

Miért jó, hogy **sed** és **grep** is van, hiszen a **grep** a **sed** egy egyszerű speciális esete? A válasz részben történeti – a **grep** jóval régebbi, mint a **sed**. A **grep** azonban túlélte a megrázkódtatást, és még gyarapodhatott is, mert a **sed** paranccsal is megoldható feladatokra egyszerűbb a használata: a **grep** olyan tömören oldja meg a feladatot, amennyire csak lehet. (Ezenkívül néhány olyan dolgot is tud, amit a **sed** nem; nézzük meg pl. a **-b** opciót!) Ám nem minden program él örökké. Volt egyszer egy **gres** nevű program, amely egyszerű helyettesítést hajtott végre, de amikor a **sed** megjelent, szinte rögtön eltűnt.

Újsor-karaktereket a **sed**-del ugyanazzal a szintaxissal lehet beszúrni, mint az **ed**-del:

```
$ sed 's/$/\
> /'
```

minden sor után új sort illeszt be, ezáltal a kimeneten dupla sorköz lesz. A

```
$ sed 's/[ →][ →]*/\
> /g'
```

minden szóköz és tabulátor helyett újsor-karaktert ír, így a bemenetet egyszavas sorokra tagolja. (A *[→]* szabályos kifejezés szóközt vagy tabulátort egyeztet, a *[→]** pedig ezekből nullát vagy többet egyeztet, így a teljes minta egy vagy több szóköz és/vagy tabulátor esetén mutat egyezést.)

Szabályos kifejezéssel vagy sorszámpárokkal kiválaszthatjuk azt a sortartományt, amelyen belül a parancsok működni fognak:

```
$ sed -n '20,30p'          Csak a 20 és 30 közötti sorokat írja ki.
$ sed '1,10d'              1-től 10-ig törli a sorokat ( = tail +11).
$ sed '1,/^$/d'            Az első üres sorig, azt is beleértve mindent töröl
$ sed -n '/^$/,/^end/p'    Kiírja az üres sor után a következő end kezdetű sorig terjedő sorok
                           csoportjait.
$ sed '$d'                 Törli az utolsó sort.
```

A sorszámozás a bemenet kezdetekor indul; új fájl elején nem kezdődik elölről.

A **sed**-nek van egy alapvető korlátja, amely az **ed**-nek nincs: nem lehet relatív sorszámokat megadni. A sorszám kifejezésekben a + és – nem szerepelhet, így nem lehet a bemeneten visszafelé haladni.

```
$ sed '$-1d'               Szabálytalan: nem lehet visszafelé hivatkozni!
Unrecognized command: $-1d
$
```

Ha egy sort beolvas, az előtte levő örökre eltűnik, az utolsó előtti sort nem lehet azonosítani, a fenti parancshoz pedig ez kellene. (A teljességhez hozzátartozik, hogy lehet ezt a **sed**-del kezelni, de nagyon nehéz. Nézzük meg a kézikönyvben a **hold** parancsot!)

Relatív előrecímzést sem lehet használni:

```
$ sed '/dolog/+1d'          Szabálytalan: nem lehet előre hivatkozni!
```

A **sed** paranccsal egyszerre több kimeneti fájlt is írhatunk. Például:

```
$ sed -n '/minta/w file1
>        /minta/!w file2' fájlnevek ...
$
```

a *minta* mintával egyező sorokat a **file1**-be, a mintával nem egyező sorokat a **file2**-be írja. Vagy, vegyük elő újra első példánkat:

```
$ sed 's/UNIX/UNIX(TM)/gw u.out' fájlnevek ... >output
```

a teljes kimenetet kiírja az **output** nevű fájlba úgy, mint korábban, de a megváltoztatott sorokat az **u.out** fájlba is kiírja.

Néha a burokkal együttműködve kaphatjuk meg a **sed** parancs közepébe helyezendő argumentumot. Példa erre az **újabb** nevű program, ami kiírja egy jegyzék azon fájljait, amelyek a megadottnál újabbak:

```
$ cat újabb
# újabb f: f-nél újabb fájlok listája
ls -t | sed '/^'$1'$/q"
$
```

Az idézőjelek védik a **sed**-nek szánt különleges karaktereket, de a *$1*-et szabadon hagyják, hogy a burok behelyettesítse a fájlnévvel.

Az argumentum így is írható:

```
"/^$1\$/q"
```

mert a *$1* helyére kerül az argumentum, és a *\$*-ból éppen *$* lesz.

Hasonlóan megírhatjuk a **régebbi** nevű programot, amely kiírja a megnevezettnél régebbi fájlokat:

```
$ cat régebbi
# régebbi f: f-nél régebbi fájlok listája
    ls -tr sed | '/^$'1'$/q'
    $
```

Az egyetlen különbség az **ls** parancs **-r** opciója, amely a sorrendet megfordítja.

Bár a **sed** sokkal többet tud, mint amit eddig megmutattunk (többek között tesztfeltételeket, ciklusszervezést, elágaztatást, korábbi sorok megjegyzését és az 1.

A **sed** parancsok összefoglalása

a\	hozzáfűz a kimenethez sorokat, amíg valamelyik nem \-re végződik
b *cimke*	ugrás a : *cimke* parancsra
c\	cserélje a sorokat a következő szövegre, mint a
d	törölje a sort; vegye a következő bemenő sort
i\	szúrja be a következő szöveget a következő kimenet elé
l	írja ki a sort; minden nem látható karaktert tegyen láthatóvá
p	írja ki a sort
q	lépjen ki
r *fájl*	olvassa a *fájl*-t és másolja tartalmát a kimenetre
s/*régi*/*új*/*f*	helyettesítse az *új*-at a *régi* helyére. Ha *f*=g minden előfordulást cseréljen ki; *f*=p írjon ki; *f*=w *fájl* a *fájl*-ba írjon
t *cimke*	vizsgálat: címkére ugrik, ha az aktuális sorban volt helyettesítés
w *fájl*	a sort a *fájl*-ba írja
y/*str1*/*str2*/	minden *str1* karaktert a megfelelő *str2* karakterre cserél (tartomány nem megengedett)
=	kiírja az aktuális bemenő sor számát
!*cmd*	hajtsa végre a **sed** *cmd* utasítását, ha a sor nincs kiválasztva
: *cimke*	címke a b és t parancsoknak
{	a hozzá tartozó } jelig egy csoportnak tekinti a parancsokat

függelékben ismertetett **ed** parancsok nagy részét), legtöbbször mégis a fenti példák szerint használjuk – egy-két egyszerű szerkesztésre –, és nem írunk vele hosszú bonyolult szerkezeteket. A 4.2. táblázat összefoglalja a **sed** egysoros lehetőségeit.

A **sed** kényelmes, mert tetszőleges hosszúságú bemeneteket képes kezelni, mert gyors, és mert a szabályos kifejezések és a soronkénti feldolgozás nagyon hasonlít az **ed**-éhez. Hátránya, hogy a **sed** nagyon korlátozott tárkapacitású (nagyon nehezen jegyez meg sorokat), csak egyszer fut végig az adatokon, nem lehet visszafelé menni, nem lehet (pl. /.../+*1* formában) előrehivatkozni, és nem lehet vele számokat kezelni – tisztán szövegszerkesztő program.

4-5. feladat. Módosítsuk **régebbi** és **újabb** nevű programjainkat úgy, hogy ne tartalmazzák kimenetükön az argumentumfájlt! Változtassuk meg őket úgy, hogy a fájlokat fordított sorrendben írják ki! □

4-6. feladat. A **sed** felhasználásával írjuk át hatékonyabbra a **bundle** programot! Útmutatás: az **itt_a_dokumentum**-okban a végjelző szót a teljes sor azonosítása alapján ismerjük fel. □

4.4. Az awk mintakereső és -feldolgozó nyelv

A **sed** bizonyos korlátait az **awk** oldja fel. Az **awk** alapgondolata hasonlít a **sed**-éhez, a részletmegoldások azonban inkább a C programnyelvre, mint egy szövegszerkesztő programra támaszkodnak. Használata ugyanolyan, mint a **sed**-é;

```
$ awk 'program' fájlnevek ...
```

de a program különbözik:

```
minta { tevékenység }
minta { tevékenység }
...
```

Az **awk** a **fájlnevekben** levő bemenetet soronként beolvassa. Minden sort minden mintá-val összehasonlít. A sorral egyező mintára végrehajtja a hozzá tartozó tevékenységet. Ugyanúgy, mint a **sed,** az **awk** sem változtatja meg a bemeneti fájlokat.

A minták lehetnek olyan szabályos kifejezések, mint az **egrep**-nél, vagy lehetnek bonyolultabb, C nyelvre emlékeztető feltételek. Egy egyszerű példa:

```
$ awk '/szabályos kifejezés/ { print }' fájlnevek ...
```

Ez ugyanazt csinálja, mint az **egrep:** kiírja a szabályos kifejezéssel egyezést mutató sorokat.

A minta is, a tevékenység is opcionális. Ha a tevékenységet elhagyjuk, az alapértelmezés szerinti tevékenység az egyező sorok kiírása, tehát a

```
$ awk '/szabályos kifejezés/' fájlnevek ...
```

ugyanazt csinálja, mint az előző példa. Megfordítva, ha a mintát hagyjuk el, akkor az **awk** a tevékenységet *minden* bemenő sorra végrehajtja. Így

```
$ awk '{ print }' fájlnevek ...
```

ugyanazt csinálja mint a **cat,** jóllehet sokkal lassabb.

Még egy utolsó megjegyzés az érdekesebb példák előtt. A **sed** parancshoz hasonlóan az **awk** parancsnak is megadhatjuk a *programot* egy fájlból:

```
$ awk -f parancsfájl fájlnevek ...
```

Mezők

Az **awk** minden bemenő sort automatikusan *mezőkre*, azaz szóközökkel vagy tabulátorokkal elválasztott, szóközt nem tartalmazó karakterláncokra oszt. E definíció szerint a **who** kimenete öt mezőből áll:

```
$ who
you    tty2   Sep 29 11 : 53
jim    tty4   Sep 29 11 : 27
$
```

Az **awk** a mezőket $1, $, ...,$NF nevekkel jelöli, az **NF** változó értéke a mezők száma. Fenti példánkban az **NF** értéke mindkét sornál 5. (Vegyük észre a különbséget: **NF** a mezők száma és a **$NF** a sor utolsó mezője. Az **awk** esetében, a burokkal ellentétben csak a mezők kezdődnek *$* jellel, a változóknak nincs ilyen ékessége.) Például ha nem akarjuk kiíratni a **du −a** parancs által szolgáltatott fájlméreteket, akkor írjuk a következő parancsot:

```
$ du −a / awk '{ print $2 }'
```

Ha a bejelentkezett személyeket és a bejelentkezés idejét akarjuk kiíratni úgy, hogy minden sorba egy kerüljön, akkor ezt így tehetjük:

```
$ who / awk '{ print $1, $5 }'
you    11 : 53
jim    11 : 27
$
```

Idő szerint rendezve ugyanez:

```
$ who / awk '{ print $5, $1 }' / sort
11 : 27   jim
11 : 53   you
$
```

Bár az **awk** használata a **sed**-énél egyszerűbb, a működése és az indítása általában lassabb. Ha sok bemenet van, akkor a végrehajtása is lassabb.

Az **awk** az üres helyeket (szóközöket és tabulátorokat) általában mezőelválasztóként értelmezi, de meg lehet adni, hogy az elválasztó bármely más, de csak egy karakter legyen. Ennek egyik módja a **−F** (nagybetűs) parancssoropció. Például a **/etc/passwd** jelszófájlban a mezők kettősponttal vannak elválasztva:

```
$ sed 3q /etc/passwd
root:3D.fHR5KoB.3s:0:1:S.User:/:
ken:y.68wd1.ijayz:6:1:K.Thompson:/usr/ken:
dmr:z4u3dJWbg7wCk:7:1:D.M.Ritchie:/usr/dmr:
$
```

Az első mezőben levő felhasználóneveket így írathatjuk ki:

```
$ sed 3q /etc/passwd | awk -F: '{ print $1 }'
root
ken
dmr
$
```

A szóközöket és a tabulátorokat szándékosan tekintjük speciális karakternek. Alapértelmezésben a szóköz és a tabulátor egyaránt elválasztó karakter, és a sor elejei elválasztók nem számítanak. Ha elválasztóként nem szóközt adunk meg, akkor a mezők számlálásakor a sor elejei elválasztókat is figyelembe veszi. Ha éppen a tabulátor az elválasztó, akkor a szóközök nem elválasztók, a sorelejei szóközök ilyenkor a mező részei és minden tabulátor egy mezőt definiál.

Kiírás

Az **awk** a bemeneti mezők számán kívül más érdekes mennyiségeket is nyomon követ. Az **NR** beépített változó az aktuális bemeneti „rekord" vagy sor sorszámát adja meg. Ezért egy bemenő adatáradathoz így adhatunk sorszámokat:

```
$ awk '{ print NR, $0 }'
```

A **$0** a teljes bemenő sor, változatlanul.

A **print** utasításban vesszőkkel elválasztott tételek a kimenetimező-elválasztó karakter, alapértelmezésben szóköz által elválasztva kerülnek kiírásra.

A **print** nyomtatási formátuma általában megfelelő, ha mégsem, akkor a **printf** utasítással tetszés szerint vezérelhetjük a kimenetet. Például ha a sorszámokat négyjegyű mezőben akarjuk kiíratni, akkor így járhatunk el:

```
$ awk '{ printf "%4d %s\n", NR, $0 }'
```

a *%4d* négyjegyű decimális egész számot (**NR**) ad meg, a *%s* egy karakterlánc (**$0**) és a *\n* az újsor-karakter, a **printf** ugyanis nem ír automatikusan szóközöket vagy új sorokat. A **awk** program **printf** utasítása olyan, mint a C nyelv ilyen nevű függvénye; l. printf(3).

Az **ind** első változatát (l. fejezetünk elejét) most így is megírhatjuk:

```
awk '{ printf "\ t%s\n", $0 }' $*
```

ami kiír egy tabulátort (*\t*), majd a bemeneti rekordot.

Minták

Tegyük fel, hogy meg akarjuk nézni az **/etc/passwd** fájlban, kiknek nincs jelszavuk. A rejtjelezett jelszó a második mező, tehát a program csak egyetlen mintából áll:

```
$ awk -F: '$2 == ""' /etc/passwd
```

A minta megkérdezi, hogy a második mező üres-e (a == az egyenlőségvizsgáló operátor). Ezt a mintát többféleképpen megadhatjuk:

`$2 == ""`	A második mező üres.
`$2 ~ /^$/`	A második mező megegyezik az üres karakterlánccal.
`$2 !~ /./`	A második mező semmilyen karakterrel sem egyezik meg.
`length($2) == 0`	A második mező hossza nulla.

A ~ jel a szabályos kifejezés egyezését, a !~ jel pedig a különbözőségét jelzi. Magát a szabályos kifejezést törtvonalak közé tesszük.

A **length** az **awk** egy beépített függvénye, amely a karakterlánc hosszát adja meg. A minta a ! jellel tagadható mint pl.:

```
!($2 == "")
```

A ! operátor a minta előtt áll úgy, mint a C nyelvben, ellentétben a **sed**-del, ahol a ! jel követi a mintát.

Az **awk** minták gyakori alkalmazása az egyszerű adatellenőrzés. Sokszor csak annyit vizsgálunk, hogy a sorok valamilyen feltételt kielégítenek-e, és ha nincs kimenet, az adatot elfogadjuk. A következő minta például csak azt ellenőrzi, hogy minden bemeneti rekordnak páros számú mezője van-e, a maradékot a % operátorral számolja:

```
NF % 2 != 0     #nyomtat, ha páratlan számú mező van
```

A következő a **length** beépített függvény segítségével kijelzi a túl hosszú sorokat:

```
length($0) > 72  #nyomtat, ha túl hosszú
```

Az **awk** ugyanúgy jelöli a megjegyzést, mint a burok; a # jel jelzi a megjegyzés kezdetét.

Több információt kaphatunk, ha kiírunk egy figyelmeztetést, és egy másik beépített függvénnyel, a **substr**-rel a túl hosszú sor elejét.

```
length($0) > 72 { print NR, "sor túl hosszú:", substr($0,1,60) }
```

A **substr(s,m,n)** függvény az **s** mezőben levő karakterláncnak az **m**-edik karakterénél kezdődő, **n** darab karakterből álló részét írja ki. (A karakterlánc az 1. pozíciónál kezdődik.) Ha az **n**-et elhagyjuk, akkor az **m**-ediktől a végéig írja ki a karaktereket. A **substr**-t rögzített helyzetű mezők kiemelésére is használhatjuk, pl. a **date** kimenetéből kiválasztjuk az órát és percet:

```
$ date
Thu Sep 29 12 : 17 : 01 EDT 1983
```

```
$ date | awk '{ print substr($4, 1, 5) }'
12 : 17
$
```

4-7. feladat. Hány olyan **awk** programot tudunk írni, amely a bemenetet a kimenetre másolja ugyanúgy, mint a **cat**? Melyik a legrövidebb? □

A BEGIN és END minták

Az **awk** két különleges mintája a *BEGIN* és az *END*. A *BEGIN* tevékenységeket az **awk** parancs az első bemenő sor beolvasása előtt hajtja végre; a *BEGIN*-t változók inicializálására, fejléc nyomtatására, vagy az **FS** változónak értéket adva a mezőelválasztó kijelölésére használhatjuk:

```
$ awk 'BEGIN { FS = ":" }
>       $2 == "" ' /etc/passwd
$                                   Nincs kimenet: mindenkinek van jelszava.
```

Az *END* tevékenységeket az **awk** a bemenet utolsó sorának feldolgozása után hajtja végre:

```
$ awk 'END { print NR }' ...
```

kiírja a bemenet sorainak számát.

Aritmetika, változók

Eddigi példáinkban egyszerű szövegkezeléssel foglalkoztunk. Az **awk** igazi ereje abban rejlik, hogy a bemenő adatokkal számolni is tud: az adatokat egyszerűen megszámlálhatja, összeget, átlagot és hasonló dolgokat számolhat. Egy gyakori **awk**-alkalmazás a számoszlopok összeadása. Például az első oszlop összes számát így adhatjuk össze:

```
        {s = s + $1 }
END     { print s }
```

Az adatok száma az **NR** változóban hozzáférhető, ezért az utolsó sort megváltoztatva:

```
END     { print s, s/NR }
```

az összeget és az átlagot is kiírhatjuk.

Példánk mutatja, hogyan használjuk a változókat az **awk**-ban. Az **s** nem beépített változó, azzal definiáljuk, hogy használni kezdjük. Alapértelmezés szerint a változók kezdőértéke nulla, ezért általában nem kell törődnünk az inicializálásukkal.

Az **awk** a C nyelvben levő aritmetikaioperátor-rövidítéseket is ismeri, ezért példánkat így is írhatjuk:

```
        { s += $1 }
END     { print s }
```

Az *s += $1* ugyanazt jelenti, mint az *s = s + $1*, csak tömörebb.

Bemenő sorokat számláló példánkat az alábbi módon általánosíthatjuk:

```
        { nc += length($0) + 1      # karakterek száma, 1 a \n miatt kell
        nw += NF                    # szavak száma
        }
END     { print NR, nw, nc }
```

Ez a bemenetén megszámolja a sorokat, szavakat és karaktereket, vagyis ugyanazt csinálja, mint a **wc** (bár fájlonként külön nem adja meg a végeredményeket).

Következő aritmetikai példánk egy olyan program, amely megadja a **pr** bemenetére érkező fájlokból kinyomtatott 66 soros lapok számát. Ezt beírhatjuk egy parancsba, legyen annak neve **prlapok:**

```
$ cat prlapok
# prlapok: kiszámolja a pr által kiírandó lapok számát
wc $* |
awk '!/ total$/ { n += int(($1+55) / 56) }
    END         { print n }'
$
```

A **pr** 56 sornyi szöveget ír minden lapra (gyakorlati tapasztalat). A **wc** kimenet minden olyan sorára, amelynek a vége nem egyezik a total-lal, a lapok számát felfelé kerekíti, majd az **int** beépített függvénnyel az egészrészt veszi.

```
$ wc sz4.*
        753      3090     18129    sz4.1
        612      2421     13242    sz4.2
        637      2462     13455    sz4.3
        802      2986     16904    sz4.4
         50       213      1117    sz4.9
       2854     11172     62847    total
$ prlapok sz4.*
53
$
```

Az eredményt ellenőrizhetjük, ha a **pr**-t közvetlenül az **awk**-ba futtatjuk:

```
$ pr sz4.* | awk 'END { print NR/66 }'
53
$
```

Az **awk** változói karakterláncokat is tárolnak. Egy változó számként vagy karakterláncként való kezelése a szövegösszefüggéstől függ. Egy s+=$1 jellegű aritmetikai kifejezésben a számértéket, egy x="abc" jellegű karakterlánchoz kapcsolódó összefüggésben a karakterértéket, kétértelmű esetben, mint pl. x>y karakterértéket használ a program, kivéve ha az operandusok nyilvánvalóan numerikusak. (Az **awk** kézikönyv pontosan leírja a szabályokat.) A karakteres változók kezdőértéke az üres karakterlánc. A következő szakaszokban jó hasznát vesszük majd a karakterláncoknak.

4.3. táblázat

Az **awk** beépített változói

FILENAME	az aktuális bemeneti fájl neve
FS	mezőelválasztó karakter (alapértelmezés: szóköz és tabulátor)
NF	a bemeneti rekord mezőszáma
NR	a bemeneti rekordok száma
OFMT	a kimenő számok formátuma (alapértelmezés: **%g**; l. **printf(3)**)
OFS	kimenő mezőelválasztó karakterlánc (alapértelmezés: szóköz)
ORS	kimeneti rekordelválasztó karakterlánc (alapértelmezés: újsor)
RS	bemeneti rekordelválasztó karakter (alapértelmezés: újsor)

4.4. táblázat

Az **awk** operátorok (növekvő precedenciarendben)

= += -= *= /= %=	értékadás; v *op*= *kif* ugyanaz, mint v = v *op* (*kif*)
||	VAGY: *kif1* || *kif2* igaz, ha az egyik vagy a másik igaz; *kif2*-t nem értékeli, ha *kif1* igaz
&&	ÉS: *kif1* && *kif2* igaz, ha mindkettő igaz; *kif2*-t nem értékeli, ha *kif1* hamis
!	a kifejezés értékének tagadása
> >= < <= == !=	relációs operátorok; ~ az egyezés, !~ a nemegyezés
~ !~	
semmi	karakterlánc-összekapcsolás
+ -	összeadás, kivonás
* / %	szorzás, osztás, maradék
++ --	növelés, csökkentés (elöl vagy hátul)

Az **awk** mindkét változótípusból több beépített változót is használ, ilyenek az **NR** és az **FS**. A beépített változók teljes listáját a 4.3. táblázat, az operátorok listáját a 4.4. táblázat mutatja be.

4-8. feladat. A **prlapok**-at ellenőrző tesztünk további megoldásokat sugall. Próbáljuk ki, melyik a leggyorsabb! □

Vezérlés-ellenőrzés

Nagyon könnyű (tapasztalatból tudjuk) egy nagyobb szöveg szerkesztésekor szomszédos szavakat véletlenül ismételni, és természetes, hogy ez szinte sosem szándékosan történik. Ez ellen véd a Writer's Workbench programcsalád egyik tagja, a **double**, amely szomszédos azonos szópárokat keres. A következő példában bemutatjuk a **double** egy **awk** parancsra épülő megvalósítását:

```
$ cat double
awk '
FILENAME != prevfile {      # új fájl
    NR = 1                  # sorszámtörlés
    prevfile = FILENAME
}
NF > 0 {
      if ($1 == lastword)
        printf "double %s, file %s, line %d\n",$1,FILENAME,NR
      for (i = 2; i <= NF; i++)
      if ($i == $(i-1))
        printf "double %s, file %s, line %d\n",$i,FILENAME,NR
      if (NF > 0)
        lastword = $NF
}' $*
$
```

A ++ operátor az operandust növeli, a -- operátor csökkenti eggyel.

A beépített **FILENAME** változó az aktuális bemeneti fájlnevet tartalmazza. Miután az **NR** a bemenet kezdetétől számolja a sorokat, minden alkalommal nullázzuk, ha a fájlnév megváltozik, így a sorokat pontosan azonosítani lehet.

Az **if** utasítás pontosan ugyanolyan, mint a C nyelvben:

```
if (feltétel)
        utasítás1
else
        utasítás2
```

Ha a *feltétel* igaz, az *utasítás1* következik; ha hamis és van **else** ág, akkor az *utasítás2* következik. Az **else** ág elmaradhat.

A **for** utasítás ugyanolyan ciklus, mint a C nyelvben, de különbözik a burok **for** ciklusától:

```
for (kifejezés1; feltétel; kifejezés2)
        utasítás
```

A **for** azonos a következő **while** utasítással, amely szintén létezik az **awk**-ban:

```
kifejezés1
while (feltétel) {
        utasítás
        kifejezés2
}
```

Például, a

```
for (i = 2; i <= NF; i++)
```

ciklusban az i végigfut a 2, 3, ... értékeken egészen a mezők **NF** számáig.

A **break** utasítás az őt közrefogó **while** vagy **or** ciklusból történő azonnali kilépést okoz; a **continue** utasítás hatására (a **while** *feltétel*-évei, a **for** *kifejezés2*-jénél) elkezdi a következő iterációt. A **next** utasítás hatására beolvassa a következő bemenő sort és az **awk** program újra elölről kezdi a minta egyezésének vizsgálatát. Az **exit** utasítás hatására a vezérlés azonnal az *END* mintára kerül.

Tömbök

Az **awk**-ban is, mint a legtöbb programnyelvben, vannak *tömbök*. Egyszerű példaként az alábbi **awk** program a bemenetének minden sorát a sorszámmal indexelve külön tömbelembe gyűjti, azután fordított sorrendben kiírja az elemeket:

```
$ cat fordítva
# fordítva: a bemenetet fordított sorrendben kiírja
awk' { line [NR] = $0 }
END  { for (i = NR; i > 0; i--) print line[i] } ' $*
$
```

Vegyük észre, hogy a tömböket, ugyanúgy mint a változókat, nem kell előre deklarálni: egy tömb méretét csak a gépben elérhető tárterület korlátozza. Ha egy nagyon nagy fájlt olvasunk be a tömbbe, akkor természetesen előfordulhat, hogy nem férünk el a tárban. Ha egy nagy fájl végét kívánjuk kiíratni, együtt kell működnünk a **tail**-lel:

```
$ tail -5 /usr/dict/web2 | fordítva
zymurgy
zymotically
zymotic
```

```
zymosthenic
zymosis
$
```

A **tail** kihasználja a fájlrendszer *keresés* (seeking) lehetőségét: a fájl végét anélkül keresi meg, hogy a közben levő adatokat elolvasná. Nézzük meg az **lseek** leírását a 7. fejezetben! (A mi rendszerünkben a **tail**-nek van egy **−r** opciója, amely a sorokat fordított sorrendben írja ki, ez feleslegessé teszi a **fordítva** programot.)

A normál feldolgozás a bemenet minden sorát mezőkre bontja. Ugyanezt a mezőkre bontó műveletet a **split** beépített függvénnyel bármely karakterláncra alkalmazhatjuk:

```
n = split(s, tmb, elv)
```

A **split** az **s** karakterláncot mezőkre bontja és azokat a **tmb** nevű tömb 1-től *n*-ig számozott elemeiben tárolja. Ha az **elv** elválasztó karaktert megadjuk, akkor azt használja, egyébként az **FS** értékét. Pl. a *split($0,a," : ")* a kettőspontoknál osztja a bemenő sort mezőkre, ami jó pl. a **/etc/passwd** feldolgozásához; a *split("9/29/83",date,"/")* a dátumot törtvonalaknál választja mezőkre.

```
$ sed 1q /etc/passwd | awk '{split($0,a,":"); print a [1]}'
root
$ echo 9/29/83 | awk '{split($0,date,"/"); print date [3]}'
83
$
```

A 4.5. táblázat az **awk** beépített függvényeit mutatja be.

4.5. táblázat

Az **awk** beépített függvényei

cos(*kif*)	a *kif* koszinusza
exp(*kif*)	a *kif* exponenciális függvénye
getline()	a következő bemenő sort olvassa, 0-át ad vissza, ha fájl vége, 1-et, ha nem
index(*s1,s2*)	az *s2* helyzete az *s1* karakterláncban, 0-át ad vissza, ha nincs benne
int(*kif*)	*kif* egészrésze
length(*s*)	az *s* karakterlánc hossza
log(*kif*)	*kif* természetes logaritmusa
sin(*kif*)	*kif* szinusza
split(*s,a,c*)	az *s*-t *c* karakter szerint *a[1]...a[n]* mezőkre bontja, visszaadja *n*-et
sprintf(*fmt, ...*)	...-ot az *fmt* szerint formázza
substr(*s,m,n*)	az *s* karakterlánc *m*-edik karakterénél kezdődő *n*-karakteres része

Társtömbök

Az adatfeldolgozásban gyakori feladat a név–érték jellegű adatpárok értékeinek összegzése. Vagyis például a

```
Susie    400
John     100
Mary     200
Mary     300
John     100
Susie    100
Mary     100
```

bemenetből kiszámoljuk a nevekhez tartozó összegeket:

```
John     200
Mary     600
Susie    500
```

Az **awk** nagyon ügyes módszert ad erre: a *társtömböket* (associative array). Rendszerint úgy gondoljuk, hogy a tömb indexe csak egész szám lehet, az **awk**-ban azonban bármilyen értéket használhatunk tömbindexként. Így

```
        { sum[$1] += $2 }
END     { for (name in sum) print name, sum[name] }
```

egy teljes program, amely összeadja a nevekhez tartozó értékeket és kiírja az összegeket, függetlenül a bemenet rendezettségétől. Minden név (**$1**) a **sum** indexe, a végén a **for** utasítás ebben a különleges formájában végigfut a **sum** elemein és kinyomtatja őket. A **for** utasításnak ez a változata szintaktikusan a következő:

```
for (változó in tömb)
    utasítás
```

Bár látszólag ez is olyan, mint a burok **for** utasítása, valójában semmi közük sincs egymáshoz. Ez a **változó** értékét minden indexre beállítva a **tömb** indexeit szervezi ciklusba, nem pedig az egyes elemeket. A tömbelemek azonban kiszámíthatatlan sorrendben képződnek, ezért célszerű lehet rendezni őket. Az előbbi példa kimenetét egy csővel a **sort**-ra köthetjük, hogy a listát a legnagyobb értékkel rendelkező személlyel kezdje:

```
$ awk '...' | sort +1nr
```

A társtömbök tárkezelése olyan tördelősémát alkalmaz, amely biztosítja, hogy mindegyik elemhez körülbelül azonos idő alatt lehessen hozzáférni, és hogy (legalábbis közepes tömbméretekig) az idő ne függjön a tömbelemek számától.

Az asszociatív tár nagyon hatékony olyan feladatokra, mint a bemenet szavainak megszámlálása:

```
$ cat szógyak
awk ' { for (i = 1; i <= NF; i++) num[$i]++ }
END   { for (word in num) print word, num[word] }
' $*
$ szógyak sz4.* | sort +1 -nr | sed 20q | 4
the 372          .CW 345          of 220          is 185
to 175           a 167            in 109          and 100
.P1 94           .P2 94           .PP 90          $ 87
awk 87           sed 83           that 76         for 75
The 63           are 61           line 55         print 52
$
```

Az első **for** ciklus végignézi a bemenő sor összes szavát, és növeli a szó által indexelt **num** tömbelemet. (Ne tévesszük össze az **awk** *$i*-jét, azaz a bemenő sor *i*-edik mezőjét a burokváltozókkal!) A fájl végigolvasása után a második **for** ciklus kiírja a szavakat és az előfordulási számukat.

4-9. feladat. A **szógyak** kimenete tartalmaz szövegformáló parancsokat is, mint pl. a **.CW**, amely megadja a szavak kinyomtatásának a betűtípusát. Hogyan szabadulhatnánk meg az ilyen nem-szavaktól? Hogyan érhetnénk el a **tr** felhasználásával, hogy a **szógyak** a bemenet kis- vagy nagybetűitől függetlenül rendesen működjön? Hasonlítsuk össze a **szógyak** felépítését és használatát a 4.2. rész csőhálózatával, valamint az alábbival:

```
sed 's/[ →] [ →]*/\
/g' $* | sort | uniq -c | sort -nr
```

□

Karakterláncok

Bár kisebb feladatokra (mint pl. egy mező kiválasztása) a **sed** és az **awk** egyaránt alkalmasak, komolyabb, valóban programozást igénylő feladatok ellátására csak az **awk** használható. Példa erre a következő program, amely a hosszú sorokat 80-oszloposakká tördeli. Minden sort, amely 80 karakternél hosszabb, a 80-adik után elvág, egy figyelmeztető \ karaktert illeszt be, majd feldolgozza a maradékot. Az utolsó sorokat nem balra, hanem jobbra illeszti, mert a programlistázáskor – amire a **fold** parancsot a leggyakrabban használjuk – így kényelmesebb. Példánkban a sorok 80 helyett 20-karakteresek:

```
$ cat test
Egy rövid sor.
Egy kicsivel hosszabb sor.
Ez a sor még valamivel hosszabb, mint az előző.
$ fold test
Egy rövid sor.
```

```
Egy kicsivel hosszab\
            b sor.
Ez a sor még valamiv\
 el hosszabb, mint az \
              előző.
$
```

Furcsa, de a 7. kiadásban nincs olyan program, amely tabulátorokat tud beilleszteni vagy törölni, bár a System V-ben a **pr** már mindkettőt tudta. A mi **fold** programunkban a **sed** segítségével a tabulátokat szóközökké alakítjuk át, hogy az **awk** karakterszámlálása pontos legyen. Ez a sor elején levő tabulátokra jól működik (programlistákban ez a gyakori), de sor közepén levőknél nem őrzi meg az oszlopszámot.

```
# fold: hosszú sorok tördelése
sed 's/→/        /g' $* |              # a tabulátort 8 szóközzé alakítja
awk '
BEGIN {
    N = 80                            #80 oszlopba tördel
    for (i = 1; i <= N; i++)          # üres karakterláncot csinál
        blanks = blanks " "
}
{   if ((n = length($0)) <= N)
        print
    else {
        for (i = 1; n > N; n -= N) {
            printf "%s\\\n", substr($0,i,N)
            i += N,
        }
        printf "%s%s\n", substr(blanks,1,N-n), substr($0,i)
    }
} '
```

Az **awk**-ban nincs explicit karakterlánc-összefűző operátor, a karakterláncok összefüggőek, ha szomszédosak egymással. Kezdetben a **blanks** üres karakterlánc. A *BEGIN* rész ciklusa összefűzéssel egy hosszú szóközláncot hoz létre: a ciklus minden újabb végrehajtása hozzátesz egy szóközt a **blanks** végéhez. A második ciklus a bemenő sorokat részenként feldolgozza, amíg a megmaradó rész nem lesz elég rövid. Mint a C nyelvben, itt is lehet értékadó utasítást kifejezésként használni, így az

```
if ((n = length($0)) <= N)...
```

szerkezet **n**-hez az érték vizsgálata előtt hozzárendeli a bemenő sor hosszát. Figyeljük meg a zárójelezést!

4-10. feladat. Módosítsuk a **fold**-ot úgy, hogy szóköznél vagy tabulátornál törje a sorokat, ne szavak közben! Tegyük alkalmassá programunkat hosszú szavakra is! □

Tegyük fel, hogy egy olyan **field n** nevű programot akarunk írni, amely minden bemenő sor n-edik mezőjét kiírja! Tehát mondhatjuk pl. hogy:

```
$ who | field 1
```

Ezzel csak a bejelentkezési neveket írjuk ki. Az **awk** nyilván tud mező szerint válogatni, a fő gondunk az, hogyan adjuk át az n számot az **awk** programnak. Íme egy megoldás:

```
awk '{ print $'$1' }'
```

A $1 kerül átadásra (ez nincs idézőjelek között), így ez lesz az a mezőszám, amit az **awk** látni fog. Egy másik megoldásban dupla idézőjeleket használunk:

```
awk "{ print \$$1 }"
```

Ebben az esetben a burok értelmezi az argumentumot, így a \$-ból $ lesz, és *$1* helyére az *n* értéke kerül. Mi jobban szeretjük az idézőjeles megoldást, mert a fél idézőjeles megoldás az **awk** programokban általában sok külön \ jelet igényel.

Második példánk az **addup n,** amely az n-edik mező számait összeadja:

```
awk '{ s += $'$1' }
END { print s }'
```

Harmadik példánk előállítja az n oszlop mindegyikének összegét és a végösszeget:

```
awk '
BEGIN { n = '$1' }
{   for (i = 1; i <= n; i++)
        sum[i] += $i
}
END { for (i = 1; i <= n; i++) {
        printf "%6g ", sum[i]
    }
    printf "; total = %6g\n", total
} '
```

A *BEGIN* segítségével n értékét egy változóba tesszük ahelyett, hogy a program további részeit teletűzdelnénk idézőjelekkel.

A fő feladat ezeknél a példáknál annak megjegyzése, hogy éppen az idézőjelen belül vagy kívül vagyunk-e (pedig éppen ez a lényeges), de ahogy ezeket most írtuk, csak a szabványos bemenetüket tudják olvasni, nem lehet nekik *n* paraméterrel kezdődő hosszú fájlnévlistát átadni. Ehhez némi burokprogramozás kell, amivel a következő fejezetben foglalkozunk.

Utolsó példánkban társtömböket használunk. Bemutatjuk, hogyan működhetünk együtt a burokkal, és egy kis programfejlesztési ízelítőt adunk.

A feladat az, hogy a rendszer minden reggel küldjön nekünk egy, az elkövetkezendő eseményekre emlékeztető üzenetet. (Lehet, hogy már van egy ilyen, l. calendar(1). Ebben a szakaszban egy másik megközelítést mutatunk be.) Az alapszolgáltatás az lesz, hogy az adott nap eseményeit megmondja. A második lépés az lesz, hogy a következő nap eseményeit is előre jelezze. A hétvégék és ünnepek megfelelő kezelését feladatként Olvasóinkra bízzuk.

Először is: keressünk helyet a naptár tárolására! Erre a legegyszerűbbnek egy **/usr/you** alatti, pl. **calendar** nevű fájl látszik.

```
$ cat calendar
Sep 30      mama születésnapja
Oct 1       ebéd Jancsival, délben
Oct 1       megbeszélés 4-kor
$
```

Másodszor: dátum szerint kell tudni keresni a naptárban. Erre sok lehetőség van; mi az **awk**-ot fogjuk használni, mert ez a legalkalmasabb az olyan aritmetikai feladatokra, hogy a „mából" a „holnapba" jussunk, de más programok, mint a **sed** vagy az **egrep** is megfelelnének. A naptár kiválasztott sorait természetesen a **mail** fogja eljuttatni nekünk.

Harmadszor: a **calendar** fájlt minden nap, lehetőleg kora hajnalban megbízhatóan és automatikusan végig kell nézni. Ezt az 1. fejezetben röviden már említett **at** programmal végeztethetjük el.

Ha a **calendar** formátumát úgy írjuk elő, hogy minden sor a **date** által előállítottal egyező hónapnévvel és nappal kezdődjön, akkor a naptárprogram első vázlata igen egyszerű:

```
$ date
Thu Sep 29 15 : 23 : 12 EDT 1983
$ cat bin/calendar
# calendar: 1. változat – csak ma
awk <$HOME/calendar '
      BEGIN { split ("'"`date`"'", date) }
      $1 == date[2] && $2 == date[3]
' | mail $NAME
$
```

A *BEGIN* blokk egy tömbbe teszi a **date** által előállított dátumot, a tömb második és harmadik eleme a hónap és a nap. Feltételezzük, hogy a **NAME** burokváltozó a bejelentkezési nevünket tartalmazza.

Az idézőjelek figyelemreméltó sorozatára azért van szükség, hogy a dátumot az **awk** program közepén egy karakterláncban tartsuk. Egy másik, egyszerűbben érthető változat, hogy a dátumot a bemenet első soraként adjuk át:

```
$ cat bin/calendar
# calendar: 2. változat – csak ma, idézőjelek nélkül
(date; cat $HOME/calendar) |
awk '
    NR == 1  { mon = $2; day=$3 } # dátumbeállítás
    NR > 1 && $1 == mon && $2 == day # naptársorok nyomtatása
' | mail $NAME
$
```

A következő lépés, hogy a **calendar** a holnapi napot ugyanúgy megnézze, mint a mait. Általában elég, ha vesszük a mai dátumot és 1-et hozzáadunk, de a hónap végén át kell lépnünk a következő hónapra és a napot vissza kell állítanunk 1-re. És persze minden hónapban más a napok száma.

Társtömbökkel ügyesen kezelhetjük a dolgot. Vegyünk két tömböt, legyen nevük **days** és a **nextmon,** indexeik a hónapok nevei, tartalmuk a hónapok napjainak száma, ill. a következő hónap neve. Így days["Jan"] tartalma 31, nextmon["Jan"] tartalma Feb. Ahelyett, hogy egy ilyen hosszú utasítássorozatot írnánk, írhatjuk a következőket:

```
days ["Jan"] = 31; nextmon ["Jan"] = "Feb"
days ["Feb"] = 28; nextmon ["Feb"] = "Mar"
....
```

A **split** segítségével alakítjuk át az alkalmas módon lerakott adatokat a valóban szükséges adatszerkezetbe:

```
$ cat calendar
# calendar: 3. változat – ma és holnap
awk <$HOME/calendar '
BEGIN {
    x = "Jan 31 Feb 28 Mar 31 Apr 30 May 31 Jun 30 " \
        "Jul 31 Aug 31 Sep 30 Oct 31 Nov 30 Dec 31 Jan 31"
    split(x, data)
    for (i = 1; i < 24; i += 2) {
        days[data[i]] = data[i+1]
        nextmon[data[i]] = data[i+2]
    }
    split("'`date`'", date)
    mon1 = date[2]; day1 = date[3]
    mon2 = mon1; day2 = day1 + 1
    if (day1 >= days[mon1]) {
      day2 = 1
      mon2 = nextmon[mon1]
    }
}
$1 == mon1 && $2 == day1 || $1 == mon2 && $2 == day2
' | mail $NAME
$
```

Figyeljük meg, hogy a Jan kétszer szerepel az adatok között, ez az „adatbástya" egyszerűbbé teszi a december feldolgozását!

Az utolsó lépés, hogy a naptárprogram minden nap fusson. Kívánságuk csupán annyi, hogy valaki ébredjen fel minden reggel 5 óra körül és futtassa le a **calendar**-t. Ezt megtehetjük magunk is, ha nem felejtjük el (minden nap!) ezt mondani:

```
$ at 5am
calendar
ctl-d
$
```

Ez azonban nem teljesen automatikus és nem elég megbízható. A trükk az, hogy az **at**-nak nemcsak azt mondjuk: futtassa a naptárprogramot, hanem azt is, hogy a következő futtatást is ütemezze:

```
$ cat early.morning
calendar
echo early.morning | at 5am
$
```

A második sor a következő napra ütemez egy következő **at** parancsot, tehát ha egyszer elindítottuk, akkor a sorozat örökmozgó. Az **at** parancs beállítja a **PATH**-ot, a munkakatalógust, és feldolgozza a parancsok számára szükséges egyéb paramétereket, így semmi különöset sem kell csinálnunk.

4-11. feladat. Módosítsuk a **calendar** programot úgy, hogy felismerje a hétvégét: pénteken a „holnapba" beleértse a szombatot, vasárnapot és hétfőt! Módosítsuk a **calendar**-t úgy, hogy a szökőévet is tudja kezelni! Kell a **calendar**-nak ismernie az ünnepeket? Hogyan oldanánk ezt meg? □

4-12. feladat. A **calendar** programnak fel kell ismernie a sor belsejében levő dátumokat, vagy elég, ha csak a sor elején levőt látja? Mi legyen a más formában (pl. 10/1/83) alakban megadott dátumokkal? □

4-13. feladat. Miért nem használja a **calendar** a **$NAME** helyett a **getname**-t? □

4-14. feladat. Írjunk egy **rm** változatot, amely nem törli a fájlokat, hanem egy ideiglenes jegyzékbe írja őket, és egy **at** paranccsal az éjszaka közepén kitisztítja ezt a jegyzéket! □

Kitekintés

Az **awk** egy terjedelmes nyelv, összes lehetőségeit egy elfogadható méretű fejezetben lehetetlen megmutatni. Felsorolunk néhány további érdekességet, amit részletesebben érdemes megnézni a kézikönyvben:

• A **print** kimeneteének fájlokba és csövekbe irányítása: bármely **print** vagy **printf** utasítás után adhatunk > jelet és egy fájlnevet (idézőjeles karakterláncként vagy változóként); a kimenetet ebbe a fájlba küldi. Ugyanúgy, mint a buroknál, a » felülírás helyett hozzátoldást jelent. Csőbe íráskor > helyett | jelet használunk.

• Többsoros rekordok: ha az **RS** rekordelválasztót újsorra állítjuk be, akkor a bemeneti rekordokat egy üres sor elválasztja. Ebben az esetben több bemenő sort egy rekordként kezelhetünk.

• A „minta, minta" elválasztó: ugyanúgy, mint az **ed**-nél vagy a **sed**-nél, egy mintapárral meg lehet adni a sorok egy tartományát. Ez az első minta előfordulásától a második minta előfordulásáig ad egyezést. Egy egyszerű példa:

```
NR == 10, NR == 20
```

a 10-iktől a 20-ik sorig, a határokat is beleértve, ad egyezést.

4.5. Jó fájlok és jó szűrők

Bár az utóbbi néhány **awk** példa teljes program, az **awk** sok alkalmazása csupán egyszerű egy-két soros program, amely egy nagyobb csőhálózat részeként valamit szűr. Ez a legtöbb szűrőre igaz – néha a felmerült feladat egy egyszerű szűrő alkalmazásával megoldható, de általában csőhálózatban összekapcsolt szűrőkkel megoldható alproblémákra bontható le. Az eszközök ilyen használatát gyakran a UNIX programozási környezet lelkének mondják. Ez nagyon leegyszerűsített szemlélet, mindazonáltal a szűrők használata átitatja a rendszert, és érdemes megfigyelni a működését.

A UNIX programok által előállított kimenet olyan formátumban van, amit más programok bemenetként megértenek. A szűrőkkel vizsgálható fájlok szövegsorokat tartalmaznak, nincsenek bennük fejlécek, végjelek, üres sorok. Minden sor egy értelmes egység – egy fájlnév, egy szó, egy futó folyamat leírása –, úgyhogy az olyan programok, mint a **wc** vagy a **grep,** meg tudják számlálni a számunkra érdekes elemeket, tudnak közöttük név szerint keresni. Ha több információ összetartozik, akkor a fájl változatlanul soronkénti szervezésű, de a sorok szóközökkel vagy tabulátorokkal mezőkre vannak felosztva, mint az **ls** **-l** kimenete. Ha az adatok így mezőkre tagolódnak, akkor az **awk** jellegű programok könnyen tudják az információt válogatni, feldolgozni vagy átrendezni.

A szűrők azonos szerkezetűek. Mindegyik a szabványos kimenetre írja az argumentumfájlok, vagy ha nincs argumentum, a szabványos bemenet feldolgozásának eredményét. Az argumentumok mindig *bemenetet* adnak meg, sosem kimenetet, így egy parancs kimenetét mindig átadhatjuk egy csőhálózatnak. Az opcionális argumentumok (vagy nem fájlnévargumentumok, mint amilyen a **grep** minta) a fájlnevek előtt állnak. Végül a hibaüzeneteket a szűrők a szabványos hibakimenetre írják, ezért azokat nem nyelik el a csövek.

Ezek a megállapodások nagyon kevéssé befolyásolják az egyes parancsokat, de

az összes programra egyöntetűen alkalmazva egyszerűvé válik az összekapcsolás. Ezt könyvünk sok példája bizonyítja, leglátványosabban talán a 4.2. szakasz végének szószámláló példája. Ha a programok bármelyike megnevezett bemenetet vagy kimenetet követel, vagy ha a paraméterek megadása párbeszédet igényel, vagy ha fejléceket vagy végjeleket kell előállítani, akkor a csőhálózat nem működik. És persze, ha a UNIX rendszerben nem volnának csövek, akkor feladataink megoldására hagyományos értelemben vett programokat kellene írnunk. De vannak csövek és működik a csőhálózat, és ha ismerjük az eszközöket, mindent nagyon egyszerűen megírhatunk.

4-15. feladat. A **ps** magyarázó fejlécet ír ki, és az **ls −l** megadja a fájlban levő blokkok számát. Mi a magyarázat? □

Történeti és irodalmi megjegyzések

A mintakereső algoritmusokról jó áttekintést találunk Al Aho, az **egrep** szerzője „Pattern matching in strings" című tanulmányában (Proceedings of the Symposium on Formal Language Theory. Santa Barbara, 1979).

A **sed**-et az **ed** alapján Lee McMahon tervezte és valósította meg.

Az **awk**-ot sokkal kevésbé elegáns módon Al Aho, Peter Weinberger és Brian Kernighan tervezték és valósították meg. A képzelőerő hiányát mutatja az is, hogy a nyelvet a szerzők után nevezték el. Az általuk írt „AWK − a pattern scanning and processing language" című tanulmány ismerteti a nyelv tervezését (*Software − Practice and Experience*, 1978. július). Az **awk** gyökerei megtalálhatók több helyen, de határozottan jó ötleteket kapott a SNOBOL4-ből, a **sed**-ből, egy Marc Rochkind által tervezett ellenőrző nyelvből, a **yacc** és **lex** eszközökből és természetesen a C nyelvből. Valójában a C nyelv és az **awk** közti hasonlóság okoz kellemetlenségeket − a nyelv úgy néz ki, mintha C lenne, de mégsem az. Néhány szerkezet hiányzik belőle, mások ravaszul eltérnek tőle.

Doug Comer „The flat file system FFG: a database system consisting of primitives" című tanulmánya ismerteti, hogyan lehet a burok és az **awk** felhasználásával adatbázisrendszert létrehozni. (*Software − Practice and Experience*, 1982. nov.).

5. Burokprogramozás

A legtöbb felhasználó azt hiszi, hogy a burok egy interaktív parancsértelmező, pedig az valójában egy programnyelv, amelynek minden utasítása lefuttat egy parancsot. Minthogy a parancsvégrehajtásnak az interaktív és a programozási vonatkozásait egyaránt a buroknak kell kielégítenie, ezért furcsa nyelvvé vált, amelyet ráadásul története és tervezése egyaránt alakított. Az alkalmazások széles köre nyugtalanítóan részletgazdag nyelvre vezetett, de szerencsére nem kell minden apró mozzanatot tökéletesen megértenünk ahhoz, hogy a nyelvet hatékonyan használjuk. Ebben a fejezetben a burokprogramozás alapjait néhány hasznos burokprogram fejlődését bemutatva ismertetjük. Könyvünk *nem* a burok kézikönyve, az a *UNIX Programozói kézikönyv* sh(1) fejezetében található. (Sokat segíthet a fejezet olvasása közben!)

A burok viselkedésének részleteit ugyanúgy mint a legtöbb parancsét, leghamarabb kísérletezéssel ismerhetjük meg. A kézikönyv esetenként érthetetlen részleteinek tisztázására nincs alkalmasabb, mint egy jó példa. Ezért is csoportosítottuk mondandónkat példák és nem a burok tulajdonságai köré; szándékunk szerint ez a fejezet útmutató arra, hogyan kell a burkot programozásra *használni.* Beszélni fogunk arról, hogy mit tud a burok, és – ötleteink interaktív tesztelését hangsúlyozva – arról is, hogy hogyan fejlesszünk és írjunk burokprogramokat.

Ha a burokban vagy bármilyen más nyelven megírtunk egy programot, lehet hogy olyat alkottunk, amit a rendszeren dolgozó többi felhasználó is használna. De a szabványok, amiket a többiek elvárnak egy programtól, általában szigorúbbak, mint amilyeneket saját magunknak előírunk. A burokprogramozásnál a többi felhasználó miatt fontos, hogy a programok a nem megfelelő bemeneteket is kezelni tudják, és a rendellenes futásról is hasznos információt adjanak.

5.1. A cal parancs átalakítása

Burokprogramokat leggyakrabban programok felhasználói interfészének fejlesztésére, módosítására írnak. Egy ilyen fejlesztésre példaként tekintsük a **cal(1)** parancsot:

```
$ cal
usage: cal [month] year
```

Eddig jó.

```
$ cal october 1983
Bad argument
$ cal 10 1983
   October 1983
S  M Tu W Th F  S
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$
```

Nem is olyan jó.

Apró kellemetlenség, hogy a hónapot számmal kell megadni. Továbbá, mint kiderül, **cal 10** a 10. év teljes naptárát írja ki, nem pedig az adott év októberét, ahhoz tehát, hogy egy havi naptárt kapjunk, mindig meg kell adnunk az évet is.

Lényeges és érdekes lehetőség az, hogy a **cal** parancs paraméterezését megváltoztathatjuk anélkül, hogy magát a **cal**-t változtatnánk. Saját **bin** jegyzékünkben elhelyezhetünk egy parancsot, amely egy kényelmesebb argumentum szintaxist átalakít olyanná, amilyet az igazi **cal** igényel. Saját változatunkat is nevezhetjük **cal**-nak, egy dologgal kevesebb, amit meg kell jegyeznünk.

Először tervezzünk: mit is csináljon a **cal**? Alapvetően azt szeretnénk, hogy ismerje fel a hónapot a neve alapján; két argumentummal viselkedjen ugyanúgy, mint a régi **cal**, de a hónapneveket alakítsa át számokká; egy argumentummal az adott hónap aktuális évi naptárát vagy az adott év naptárát nyomtassa; argumentum nélkül az *aktuális* hónap naptárát írja ki, hiszen minden bizonnyal ez a **cal** leggyakoribb alkalmazása. Tehát a feladat mindössze annyi, hogy el kell dönteni, hány argumentum van megadva, azután le kell képezni őket a szabványos **cal** által kívánt formára.

A buroknak van egy **case** utasítása, amely nagyon alkalmas ilyen döntésekre:

```
case szó in
minta) parancsok ; ;
minta) parancsok ; ;
...
esac
```

A **case** utasítás felülről lefelé összehasonlítja a szó-t a minták-kal, és végrehajtja azokat a parancsokat, amelyek az első és csakis az első, egyezést adó mintá-hoz tartoznak. A mintákat a burok minta szerinti keresési szabályai szerint adjuk meg, amelyeket a fájlnévkeresés szabályaival egy kicsit általánosítottunk. Minden tevékenységet dupla pontosvessző (*; ;*) zár le. (Az utolsó tevékenységnél ugyan a pontosvesszők elhagyhatók, de az egyszerűbb szövegszerkesztés miatt gyakran kiírjuk.)

A saját **cal**-változatunkat tehát eldönti, hány argumentum van, feldolgozza az alfabetikus hónapneveket, majd hívja az igazi **cal**-t. A $# burokváltozó tartalmazza azt az argumentumszámot, amellyel a burokfájlt behívtuk; a többi speciális burokváltozót az 5.1. táblázatban soroljuk fel.

A burok beépített változói

$#	az argumentumok száma
$*	minden argumentumot a buroknak
$@	hasonlót a $*-hoz; l. az 5.7. szakaszt
$–	burok által adott opciók
$?	az utoljára végrehajtott parancs visszaadott értéke
$$	a burok folyamatazonosítója
$!	az utolsó, &-del indított folyamat azonosítója
$HOME	a **cd** parancs alapértelmezés szerinti argumentuma
$IFS	argumentumszavakat elválasztó karakterek listája
$MAIL	fájl, amely ha megváltozik, „you have mail" üzenetet küld
$PATH	jegyzéklista a parancsok kereséséhez
$PS1	prompt karakterlánc; alapértelmezés: $
$PS2	prompt karakterlánc parancssor folytatáshoz, alapértelmezés: >

```
$ cat cal
# cal: kellemesebb interfész a /usr/bin/cal-hoz

case $# in
0)      set `date` ; m=$2; y=$6 ; ;          # nincs argumentum: a mai nap a mai dátum
1)      m=$1; set `date` ; y=$6 ; ;          # 1 argumentum: ez az év
*)      m=$1; y=$2 ; ;                       # 2 argumentum: hónap és év
esac
case $m in
jan*|Jan*)          m=1 ; ;
feb*|Feb*)          m=2 ; ;
mar*|Mar*)          m=3 ; ;
apr*|Apr*)          m=4 ; ;
may*|May*)          m=5 ; ;
jun*|Jun*)          m=6 ; ;
jul*|Jul*)          m=7 ; ;
aug*|Aug*)          m=8 ; ;
sep*|Sep*)          m=9 ; ;
oct*|Oct*)          m=10 ; ;
nov*|Nov*)          m=11 ; ;
dec*|Dec*)          m=12 ; ;
[1-9]|10|11|12)  ; ;      # numerikus hónap
*)              y=$m; m="" ; ; # sima év
esac

/usr/bin/cal $m $y        # futtatja az eredetit
$
```

Az első **case** ellenőrzi az argumentum ($#) számát, és kiválasztja a megfelelő tevékenységet. Az első **case** záró * mintája lomtár: ha az argumentumok száma nem 0 vagy 1, akkor az utolsó esetet hajtja végre. (Mivel a **case** a mintákat sorban vizsgálja, a lomtár csak az utolsó lehet.) Ez az **m** és **y** változókban beállítja a hónapot és az évet – két argumentumot megadva **cal** programunk kezd az eredetire hasonlítani. Az első **case** utasítás néhány ravasz sora a

```
set `date`
```

parancsot tartalmazza. Ez első látásra nem nyilvánvaló, de kipróbálva könnyű rájönni, hogy mit csinál:

```
$ date
Sat Oct  1 06:05:18 EDT 1983
$ set `date`
$ echo $1
Sat
$ echo $4
06:05:20
$
```

A **set** beépített burokparancs sok mindent csinál. Argumentum nélkül a környezet változóinak értékeit mutatja (ezt a 3. fejezetben láttuk). A közönséges argumentumok állítják be a **$1, $2** stb. értékeket. Így a **set** *`date`* a **$1**-be a hét napját, a **$**-be a hónapot írja és így tovább. A **cal** első **case** utasítása tehát, ha nincs argumentum, akkor az aktuális dátumból beállítja a hónapot és az évet, ha egy argumentum van, akkor az lesz a hónap értéke, és csak az évet veszi az aktuális dátumból.

A **set** számos opciója közül a leggyakoribbak a **−v** és a **−x,** amelyek bekapcsolják a burok által éppen feldolgozott parancsok visszhangozását. A bonyolultabb burokprogramok nyomkövetésének ezek nélkülözhetetlen segédeszközei.

Hátravan még a szöveges formájú hónap átalakítása a hónap sorszámává. Ezt a második **case** utasítás végzi, ami nagyrészt saját magát magyarázza. Az egyetlen furcsaság benne a **case** utasítás mintájában levő / karakter, amely ugyanúgy, mint az **egrep**-nél, alternatívát jelöl: *nagy / kicsi* megfelel a nagy-nak és a kicsi-nek is. Természetesen az eseteket így is írhattuk volna: *[jJ]an** stb. A program a hónapneveket elfogadja csupa kisbetűvel is (a legtöbb program kisbetűs bemenetet vár) és nagybetűvel kezdődően is (a **date** ilyen formában ír). A burok minta szerinti keresési szabályait az 5.2. táblázat tartalmazza.

A második **case** utasítás két esete az egyetlen argumentumot kezeli, amely év is és hónap is lehet; emlékezzünk vissza, hogy az első **case** utasítás azt feltételezte, hogy ez a hónapot adta meg! Ha ez a szám lehet egy hónap, akkor hónapnak tekinti, egyébként feltételezi, hogy év.

Végül az utolsó sor az átalakított argumentumokkal behívja a **/usr/bin/cal**-t (azaz az igazi **cal** programot). A mi **cal**-változatunk úgy működik, ahogyan azt egy újszülött elvárhatja:

Minta szerinti egyezés szabályai a burokban

*	minden karakterlánccal egyezik, az üres karakterlánccal is
?	bármely egyetlen karakterrel egyezik
[*ccc*]	bármely, a *ccc*-ben előforduló karakterrel egyezik [a–d0–3] ugyanaz, mint [abcd0123]
"..."	pontosan a ... mintával egyezik, az idézőjelek a speciális karaktereket védik, ugyanilyen a '...' is
\c	betű szerint egyezés
a | b	csak **case** kifejezésekben vagy a-val, vagy b-vel egyezik
/	fájlnevekben csak a kifejezésben levő explicit /-rel egyezik, **case**-ben ugyanúgy egyezik, mint bármely más karakter
.	fájlnév első karaktereként csak a kifejezésben levő explicit .-tal egyezik

```
$ date
Sat Oct  1 06:09:55 EDT 1983
$ cal
   October 1983
S  M Tu W Th  F  S
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$ cal dec
   December 1983
S  M Tu W Th  F  S
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

$
```

és **cal 1984** kiírja a teljes 1984. évi naptárat.

Az átalakított **cal** program ugyanazt csinálja, mint az eredeti, csak egyszerűbben és könnyebben megjegyezhető módon. Emiatt választottuk inkább a már meglevő **cal**

parancsnevet is, és nem a **calendar**-t vagy valami nehezebben megjegyezhető rövidítést (pl. **ncal**-t). A név meghagyásának további előnye, hogy a naptár kiíratásához a felhasználóknak nem kell új reflexeket kialakítaniuk.

Mielőtt elhagyjuk a **case** utasítást, érdemes egy rövid magyarázatot fűzni ahhoz, hogy miért különböznek a burok mintakeresési szabályai az **ed** és leszármazottai mintakeresési szabályaitól. (A kellemetlen kettősség miatt két szabályrendszert kell megtanulnunk, és a feldolgozáshoz is két kódrész kell.) A különbözőség részint egyszerűen rossz választásból, megállapodások hiányából ered – pl. a régmúlt időkkel való kompatibilitáson kívül semmi sem indokolja, hogy az **ed** a pontot (.), a burok a kérdőjelet (?) használja „bármely karakterrel való egyezés" jelölésére. Néha azonos minták különböző feladatokat látnak el. Az **ed** szabályos kifejezés keresése sorban bárhol előforduló karakterláncokra vonatkozik, a ^ és a $ speciális karakterek a keresést a sor elejére vagy végére rögzítik. Fájlnevek esetében azonban a keresést alapértelmezésben a sor elejére kívánjuk rögzíteni, hiszen ez a leggyakoribb eset. Nagyon kényelmetlen volna, ha

```
$ ls *.c
```

helyett valami ilyesmit kellene írni:

```
$ ls ^?*.c$          így nem működik.
```

5-1. feladat. Ha a felhasználóknak tetszik új **cal**-változatunk, hogyan tehetjük általánosan hozzáférhetővé? Mit kell a /usr/bin-be tennünk? □

5-2. feladat. Érdemes-e rögzíteni, hogy **cal 83** az 1983-as év naptárját írja ki? Ha így volna, hogyan írathatnánk ki a 83-as év naptárját? □

5-3. feladat. Módosítsuk a **cal**-t úgy, hogy több hónapot

```
$ cal oct nov
```

esetleg a hónapok egy tartományát

```
$ cal oct – dec
```

is elfogadja! Ha most december van és *cal jan*-t kérünk, akkor az idei januárt vagy a jövő évit írja ki? Meddig érdemes a **cal** tulajdonságainak bővítését folytatni? □

5.2. Melyik parancs az igazi: which?

A saját parancsváltozatok, mint amilyen a **cal,** különböző kellemetlenségeket okozhatnak. Közülük a legnyilvánvalóbb; ha Mary-vel együtt dolgozunk, és **mary**-ként bejelentkezve gépeljük be a **cal** parancsot, akkor a szabványos változatot kapjuk, nem

az újat (kivéve persze, ha Mary az új változatot beírta saját **bin** jegyzékébe). Ez nagyon megtévesztő lehet – gondoljunk arra, hogy az eredeti **cal** hibajelzései nem sokat segítenek –, és jól mutatja az általános hibalehetőséget. Mivel a burok a parancsokat a **PATH** által meghatározott katalógusban keresi, mindig előfordulhat, hogy a parancs más változatát kapjuk, mint amelyet vártunk. Pl. ha beütünk egy parancsot, mondjuk az **echo**-t, akkor a valójában futtatásra kerülő fájl ösvényneve lehet **./echo** vagy **/bin/echo** vagy **/usr/bin/echo,** vagy a **PATH** elemeitől és a fájlok helyétől függően valami más. Nagy zavart okozhat, ha a keresési útban a vártnál korábban van egy adott nevű, de rossz viselkedésű végrehajtható fájl. Talán a leggyakoribb példa erre a később tárgyalásra kerülő **test** parancs. Ennek a nevét annyira gyakran szoktuk ideiglenes programváltozatok elnevezésére felhasználni, hogy bosszantóan gyakran hívunk be rossz **test** programot*. Jó szolgálatot tenne egy olyan parancs, amely tájékoztatna, hogy a program melyik változata fog végrehajtásra kerülni.

Egy megoldás erre, hogy a **PATH**-ban megnevezett jegyzékeket ciklusban végignézve megkeressük az adott nevű végrehajtható fájlokat. A 3. fejezetben a **for** paranccsal néztünk végig fájlneveket és argumentumokat. Most ilyen ciklusra lenne szükségünk:

```
for i in      PATH minden elemére
do            ha az adott név megtalálható az i jegyzékben, írja ki a teljes ösvénynevet
done
```

Miután bármely parancsot futtathatunk fordított félidézőjelek (`...`) között, kézenfekvő megoldás, hogy a **sed**-et végigfuttatva a **$PATH**-on a kettőspontokat szóközökké alakítjuk. Régi barátunkkal, az **echo**-val ellenőriztetve:

```
$ echo $PATH
:/usr/you/bin:/bin:/usr/bin          4 komponens.
$ echo $PATH | sed 's/:/ /g'
  /usr/you/bin /bin /usr/bin         Csak 3-at ír ki!
$ echo `echo $PATH | sed 's/:/ /g'`
/usr/you/bin /bin /usr/bin           Most is csak 3.
$
```

Tisztán látható a hiba oka! A **PATH**-ban az üres karakterlánc a pont (.) szinomimája. Így a **PATH** kettőspontjainak átírása szóközökké nem tökéletes megoldás – az üres komponensekkel kapcsolatos információ elvész. Ahhoz, hogy a jegyzékekről pontos listát készíthessünk, a **PATH** üres komponenseit ponttá kell konvertálnunk. Az üres elem lehet karakterlánc közepén, vagy bármelyik végén, ezért egy kicsit megdolgoztatott bennünket. Az összes esetre jó megoldás:

* Később látni fogjuk a burokfájloknál, ahol a **test**-et általában használjuk, hogyan lehet ezt kivédeni.

```
$ echo $PATH | sed 's/^:/.:/
>                  s/::/:.:/g
>                  s/:$/:./
>                  s/:/ /g'
. /usr/you/bin /bin /usr/bin
$
```

Írhattunk volna négy különálló **sed** parancsot is, de miután a **sed** sorrendben helyettesít, egy hívással is megnézethetünk mindent.

Ha már megvannak a **PATH**-ban levő jegyzékek, akkor a már említett **test(1)** parancs megmondja, hogy a fájl létezik-e az egyes jegyzékekben. A **test** parancs az egyik legügyetlenebb UNIX program. Pl. a **test −r fájl** ellenőrzi, hogy a **fájl** létezik-e és olvasható-e; a **test −w fájl** ellenőrzi, hogy a **fájl** létezik-e és írható-e, de az, amire éppen szükségünk volna, a **test −x**, a 7. kiadásban nincs meg (bár a System V. és egyéb változatok tartalmazzák). Meg kell elégednünk a **test −f** paranccsal, amely ellenőrzi, hogy a fájl létezik-e és nem jegyzék-e, másképp szólva, hogy közönséges fájl-e. Nézzük át saját rendszerünk kézikönyvének **test** lapját, mivel számos változat van forgalomban!

Minden parancs visszaad a buroknak egy *kilépési státust* (exit status) − egy olyan értéket, amely jelzi a történteket. A kilépési státus kis egész szám, megállapodás szerint a 0 „igazat" (a parancs sikeres lefutását) és a nemnulla „hamisat" (a parancs sikertelen futását) jelez. Megjegyezzük, hogy ez éppen ellentétes a C nyelvben használt igaz és hamis értékekkel.

Mivel sok különböző érték jelent „hamisat", a hiba okát gyakran a „hamis" kilépési státusba kódoljuk. Pl. a **grep** 0-t ad vissza, ha talált egyezést; 1-et, ha nem talál egyezést; 2-t, ha a mintában vagy a fájlnevekben hiba volt. Minden program visszaad egy státust, bár rendszerint nem vagyunk kíváncsiak annak értékére. A **test** program más, mint a többi, mert *egyedüli célja* éppen az, hogy visszaadjon egy kilépési státust, más szokásos kimenete nincs, és nem változtatja meg a fájlokat sem.

A burok az utolsó program kilépési státusát a **$?** változóban tárolja:

```
$ cmp /usr/you/.profile /usr/you/.profile
$                                  Nincs kimenet; azonosak.
$ echo $?
0                                  A nulla azt jelzi, hogy jó a futás; a fájlok azonosak.
$ cmp /usr/you/.profile /usr/mary/.profile
/usr/you/.profile /usr/mary/.profile differ: char 6, line 3
$ echo $?
1                                  A nemnulla azt jelzi, hogy a fájlok különbözőek.
$
```

Néhány parancs, mint pl. a **cmp** vagy a **grep,** a **−s** opcióval kiegészítve minden kimenetet elnyom, csak a státust állítja elő.

A burok **if** utasítása a parancsokat feltételesen egy adott parancs kilépési státusától függően futtatja:

```
if      parancs
then
        parancsok, ha az if feltétel igaz
else
        parancsok, ha az if feltétel hamis
fi
```

Fontos az újsor-karakterek elhelyezése: a **fi, then** és **else** szavakat a burok csak újsor vagy pontosvessző karakter után ismeri fel. Az **else** rész opcionális.

Az **if** utasítás mindig lefuttat egy parancsot – a feltételt –, míg a **case** utasítás közvetlenül a burokban végzi a minta szerinti keresést. Néhány UNIX változatban, pl. a System V.-ben a **test** a burok egy beépített függvénye, így az **if** és a **test** ugyanolyan gyorsan fut, mint a **case.** Ha a **test** nincs beépítve, akkor a **case** utasítások az **if** utasításoknál lassabbak, ezért nem célszerű minta szerinti kereséshez használni:

```
case "$1" in
hello) parancs
esac
```

gyorsabb, mint

```
if test "$1" = hello        Lassabb, kivéve, ha a test a burok beépített függvénye.
        parancs
fi
```

Ez az egyik oka annak, hogy a burokban néha **case** utasítást használunk olyan vizsgálatokra, amiket a legtöbb programnyelvben az **if** utasítással szoktunk elvégezni. A **case** utasítással viszont nem lehet egyszerűen eldönteni, hogy egy fájlnak van-e olvasás engedélye; erre jobb a **test** és egy **if.**

Készítsünk olyan **which** parancsot, amely megmondja, hogy egy parancsnak melyik fájl felel meg! Az első változat elemei már össze is gyűltek:

```
$ cat which
# melyik parancs: a PATH-ban melyik parancs kerül végrehajtásra, 1. változat
case $# in
0)      echo 'Usage: which command' 1>&2; exit 2
esac
for i in `echo $PATH | sed 's/^:/.:/
                            s/::/:.:/g
                            s/:$/:./
                            s/:/ /g'`
do
  if test -f $i/$1          # használjunk test -x-et, ha lehet
  then
        echo $i/$1
        exit 0              # megtalálta
  fi
done
```

```
exit 1                  # nem találta meg
$
```

Próbáljuk ki első változatunkat:

```
$ cx which              Tegyük végrehajthatóvá!
$ which which
./which
$ which ed
/bin/ed
$ mv which /usr/you/bin
$ which which
/usr/you/bin/which
$
```

A kezdő **case** utasítás csak hibaellenőrzés. Figyeljük meg az **echo** *1>&2* átirányítását, amelynek célja, hogy a hibaüzenetet ne nyelje el egy cső! A burokba beépített **exit** parancsot arra használjuk, hogy visszakapjuk a kilépési státust. Az exit 2 jelentése, hogy a parancs nem működik; az exit 1 jelentése, hogy nem találta, az exit 0 pedig az, hogy megtalálta a fájlt. Ha nincs explicit **exit** utasítás, akkor a burok kilépési státusa az utolsónak végrehajtott parancs kilépési státusa lesz.

Mi történik, ha munkajegyzékünkben van egy saját **test** nevű programunk? (Feltételezzük, hogy a **test** nem a burok beépített függvénye.)

```
$ echo 'echo hello' >test    Csináljunk egy hamis test-et!
$ cx test                    Tegyük végrehajthatóvá!
$ which which                Most próbáljuk ki a which-et!
hello                        Nem működik!
./which
$
```

További hibaellenőrzés is szükséges. Futtathatjuk a **which**-et (ha munkajegyzékünkben nincs **test**) úgy, hogy keresse meg a **test** teljes ösvénynevét és adja meg explicite. Ez azonban nem kielégítő: a **test** különböző rendszerekben különböző jegyzékekben lehet, továbbá a **which** függ a **sed**-től és az **echo**-tól is, így azok ösvénynevét is meg kell adnunk. Van egyszerűbb megoldás: a burokfájlban rögzítsük a **PATH**-ot úgy, hogy csak a **/bin** és a **/usr/bin**-ben keressen parancsokat. Természetesen a **which** parancs számára meg kell őriznünk a régi **PATH**-ot is, hogy az meghatározza a végignézendő jegyzékeket.

```
$ cat which
# melyik parancs: a PATH melyik parancsa kerül végrehajtásra, utolsó változat
opath=$PATH
PATH=/bin:/usr/bin
case $# in
0)    echo 'Usage: which command' 1>&2; exit 2
esac
```

```
for i in `echo $opath | sed 's/^:/.:/
                             s/: :/:.:/g
                             s/:$/:./
                             s/:/ /g'`
do
  if test -f $i/$1          # ez csak a /bin/test
  then                      # vagy /usr/bin/test lehet
       echo $i/$1
       exit 0               # megtalálta
  fi
done
exit 1                      # nem találta meg
$
```

A **which** most már akkor is működik, ha a keresési útban egy hamis **test** (vagy **sed** vagy **echo**) van.

```
$ ls -l test
-rwxrwxrwx 1 you   11 Oct 1 06 : 55 test   Még megvan.
$ which which
/usr/you/bin/which
$ which test
./test
$ rm test
$ which test
/bin/test
$
```

A buroknak két további parancsösszefűző operátora is van, a || és az **&&**. Ezek gyakran tömörebbek és kényelmesebbek, mint az **if** utasítás. Pl. némely **if** utasítás helyett || operátort alkalmazhatunk:

```
test -f fájlnév || echo file fájlnév does not exist
```

ugyanaz, mint

```
if test ! -f fájlnév      A ! jel a feltétel tagadása.
then
      echo file fájlnév does not exist
fi
```

A || operátornak formája ellenére semmi köze sincs a csövekhez – ez egy VAGY (OR) jelentésű feltételes operátor. Végrehajtja a || jel bal oldalán levő parancsot. Ha annak kilépési státusa nulla (sikeres), akkor a jobb oldalon álló parancsot elhagyja. Ha a bal oldal nemnullát ad (hiba), akkor végrehajtja a jobb oldali parancsot, és a teljes kifejezés értéke a jobb oldali parancs kilépési státusa lesz. Másképp szólva a || jel feltételes VAGY operátor, amely a jobb oldali parancsot nem hajtja végre, ha a bal oldali sikeres volt. A másik feltételes operátor az ÉS (AND) **&&** jelentésű akkor hajtja végre a jobb oldali parancsot, ha a bal oldali sikeres volt.

5-4. feladat. Miért nem állítja vissza a **which** kilépés előtt a **PATH**-ot **opath**-ra? □

5-5. feladat. A burok az **esac**-ot használja a **case** lezárására, a **fi**-t az **if** lezárására, vajon miért a **done**-t használja a **do** lezárására? □

5-6. feladat. Bővítsük a **which**-et egy **−a** opcióval, amely a **PATH** összes fájlját kiírja és nem lép ki már az első után! Útmutatás: *match='exit 0'*. □

5-7. feladat. Módosítsuk a **which**-et úgy, hogy a burokba beépített parancsokat (mint amilyen az **exit**) is ismerje fel! □

5-8. feladat. Módosítsuk a **which**-et úgy, hogy ellenőrizze a fájlok végrehajtási engedélyeit is! Változtassuk meg úgy, hogy hibaüzenetet adjon, ha nem találja a fájlt! □

5.3. Változások figyelése: while és until

A 3. fejezetben a **for** ciklust egyszerű iteratív programokban használtuk. A **for** rendszerint egy fájlnévkészleten, mint pl. **for i in** **.c* vagy burok program argumentumokon, mint pl. **for i in** *$**, szalad végig. A burokciklusok általánosabbak, mint amit ezek alapján gondolnánk. Nézzük meg, mit tudnak!

Három ciklusszervező utasítás van: a **for,** a **while** és az **until.** Mind ez idáig a **for** a legelterjedtebb. Ez egy szókészlet minden elemére egyszer végrehajt egy parancskészletet − a ciklus magját. A szavak leggyakrabban éppen fájlnevek. A **while** és az **until** a ciklusmagot egy parancs kilépési státusától függően hajtja végre: a ciklusmagot mindaddig ismétli, amíg a feltételparancs kilépési státusa nemnulla (a **while** esetén), ill. nulla (az **until** esetén). A **while** és az **until** csak a parancs kilépési státusának értelmezésében különböznek.

Íme az alapvető ciklusformák:

for i in *Szólista*
do
Ciklusmag **$i**-t a lista egymás utáni elemeire állítja.
done

for i
do
(Az implicit lista a burokfájl összes argumentuma, vagyis $*.)
Ciklusmag **$i**-t az egymás utáni elemekre állítja.
done

while *Parancs*
do
Ciklusmag Mindaddig végrehajtja, amíg a parancs kilépési státusa igaz.
done

until *Parancs*
do
Ciklusmag Mindaddig végrehajtja, amíg a parancs kilépési státusa hamis.
done

A **for** második formája, amelyben az üres lista a *$**-ot jelent, a leggyakoribb alkalmazás kényelmes rövidítése.

A **while**, ill. **until** ciklusban a vezérlőfeltétel parancsa bármilyen parancs lehet. A következő egyszerű példában egy **while** figyeli, hogy valaki (mondjuk Mary) bejelentkezik-e?

```
while sleep 60
do
        who | grep mary
done
```

A **sleep,** amely 60 másodpercig várakozik, mindig normálisan fut le (kivéve, ha megszakítjuk) és ezért „igaz" státust ad vissza, úgyhogy a ciklus percenként megnézi, hogy Mary bejelentkezett-e?

Ennek a változatnak a hátránya, hogy ha Mary már bejelentkezett, akkor is kell 60 másodpercet várnunk, míg ezt megtudjuk, de ezután ezt minden percben újra tudatja velünk. A ciklust kifordítva és az **until**-lel megírva az információt egyszer, késedelem nélkül kapjuk, amint Mary bejelentkezett:

```
until who | grep mary
do
        sleep 60
done
```

Ez egy érdekesebb feltétel. Ha Mary be van jelentkezve, akkor a *who | grep mary* parancssor a **who** listában kiírja az ő adatait, és „igaz" státust ad vissza. A **grep** ugyanis a visszaadott státusban azt jelzi, hogy talált-e valamit, és az utolsó elem kilépési státusa egyben a csőhálózat kilépési státusa.

Végül rakjuk össze ezt a parancsot, adjunk neki nevet, és építsük be a rendszerbe:

```
$ cat watchfor
# watchfor: figyeli, hogy valaki bejelentkezik-e
PATH=/bin:/usr/bin
case $# in
0)      echo 'Usage: watchfor person' 1>&2; exit 1
esac
until who | egrep "$1"
do
        sleep 60
done
$ cx watchfor
$ watchfor you
you     tty0   Oct 1 08 : 01          Működik.
$ mv watchfor /usr/you/bin             Beépítjük.
$
```

A **grep**-et **egrep**-re cseréltük, hogy ilyet is írhassunk:

```
$ watchfor 'joe|mary'
```

vagyis több személyt is kereshessünk egyszerre.

Bonyolultabb példaként figyeljük *mindenki* be- és kijelentkezését, és jelezzük az érkező és távozó személyeket – vagyis fejlesszük tovább a **who** parancsot. Az alapszerkezet egyszerű: percenként lefuttatjuk a **who**-t, kimenetét összehasonlítjuk az egy perccel korábbival, és jelezzük a különbségeket. A **who** kimenetét egy fájlban fogjuk tárolni, tegyük azt a **/tmp** jegyzékbe. Fájljainkat úgy fogjuk a más folyamatokhoz tartozóktól megkülönböztetni, hogy a **$$** burokváltozót (a burokparancs folyamatazonosítóját) beépítjük a fájlnévbe; ez egyébként elég általános fogás. A parancsnevet elsősorban a rendszeradminisztrátor kedvéért kódoljuk az átmeneti fájlokba. A parancsok (beleértve a **watchwho** mostani változatát is) gyakran hagynak a **/tmp**-ben fájlokat, jó tudni, hogy melyik parancstól származnak.

```
$ cat watchwho
# watchwho: figyeli, hogy ki jelentkezik be és ki
PATH=/bin:/usr/bin
new=/tmp/wwho1.$$
old=/tmp/wwho2.$$
>$old       # létrehoz egy üres fájlt
while :     # végtelen ciklus
do
  who >$new
  diff $old $new
  mv $new $old
  sleep 60
done | awk '/>/ { $1 = "in:  "; print }
            /</ { $1 = "out:  "; print }'
$
```

A kettőspont (**:**) a burok egy beépített parancsa és semmit sem csinál, csupán kiértékeli az argumentumait és *igaz* státust ad vissza. Használhattuk volna helyette a **true** parancsot, ami egyszerűen „igaz" kilépési státust ad. (Van egy *false* parancs is.) A kettőspont azonban hatékonyabb, mint a **true**, mert nem futtat a fájlrendszerből parancsot.

A **diff** kimenete a két fájlból jövő adatokat a < és > jelekkel különbözteti meg; az **awk** program ezt a változásjelentést könnyebben érthető formájúra alakítja. Figyeljük meg, hogy a teljes **while** ciklust az **awk**-ra csöveztük, nem indítunk percenként egy-egy új **awk**-ot! A **sed** erre a feldolgozásra nem alkalmas, mert kimenete mindig egy sorral a bemenete mögött van. Mindig van egy bemenő sor, amit már feldolgozott, de még nem írt ki, és ez felesleges késleltetést okozna.

Mivel az **old** fájlt üresfájlként hozzuk létre, ezért a **watchwho** első kimenete az összes bejelentkezett listáját adja. Megváltoztathatjuk a parancsot úgy, hogy ne üresként, hanem a **who >$old** paranccsal hozza létre az **old**-ot, úgy a **watchwho** csak a változásokat fogja kiírni – de ez már ízlés dolga.

Következő ciklikus programpéldánk figyeli a postaládánkat, és ha változást lát, kiírja hogy: Levele van. Ez hasznos változata a burok MAIL változót használó

beépített mechanizmusának. Példánkban fájlok helyett csak azért használunk burokváltozókat, hogy bemutassuk: mindent lehet másképp is csinálni!

```
$ cat checkmail
# checkmail: figyeli a postaláda növekedését
PATH=/bin:/usr/bin
MAIL=/usr/spool/mail/`getname` # rendszerfüggő
t=${1-60}
x="`ls -l $MAIL`"
while :
do
  y="`ls -l $MAIL`"
  echo $x $y
  x="$y"
  sleep $t
done | awk '$4 < $12 { print "Levele van" }'
$
```

Most azért az awk-ot használtuk, hogy biztosítsuk, ne minden változáskor, csak a postaláda növekedésekor adjon üzenetet. Különben akkor is kapnánk üzenetet, ha törölnénk valamit. (A burokban beépített változatnak megvan ez a hiányossága.)

Az időintervallum alapértelmezésben 60 másodperc, de ha a parancssorban megadunk egy paramétert:

```
$ checkmail 30
```

akkor az (30 másodperc) lesz a várakozási idő. A t burokváltozó értékét a

```
t=${1-60}
```

sor a megadott időértékre, ill. ha nem adunk értéket, 60-ra állítja. Ezzel a burok egy újabb tulajdonságához érkeztünk.

A ${var} ugyanaz, mint $var, és ezt felhasználhatjuk a karakterláncokon belüli betűket vagy számokat tartalmazó változókkal kapcsolatos gondok elkerülésére:

```
$ var=hello
$ varx=goodbye
$ echo $var
hello
$ echo $varx
goodbye
$ echo ${var}x
hellox
$
```

A zárójeleken belül bizonyos karakterek a változó különleges kezelését írják elő. Ha a változó nincs megadva, és neve után kérdőjel áll, akkor a burok kiírja a ? utáni karakterláncot, majd kilép (hacsak nem interaktív). Ha nincs hibaüzenet, akkor egy szabványos üzenetet ír ki:

```
$ echo ${var?}
hello                          Rendben: a var-nak van értéke.
$ echo ${vacak?}
vacak: parameter not set       Alapértelmezés szerinti üzenet.
$ echo ${vacak?hiba!}
vacak: hiba!                   Van üzenet.
$
```

Figyeljük meg, hogy a burok által generált üzenet mindig tartalmazza a definiálatlan változó nevét! Egy másik forma a *${var-dolog}*, amelynek értéke **$var**, ha az meg van adva és **dolog** ha nincs megadva. A *${var=dolog} jelentése hasonló, de* ***$var*** *értékét* ***dolog****-ra állítja:*

```
$ echo ${vacak-'Itt vagyok'}
Itt vagyok
$ echo ${vacak?}
vacak: parameter not set       A vacak változatlan.
$ echo ${vacak='Itt vagyok'}
Itt vagyok
$ echo ${vacak?}
Itt vagyok                     A vacak értéke: Itt vagyok:
$
```

A változók értékadására vonatkozó szabályok az 5.3. táblázatban találhatók.

Visszatérve eredeti példánkra,

```
t=${1-60}
```

a **t** értékét $1-re, vagy ha nincs argumentum, akkor 60-ra állítja.

5-9. feladat. Nézzük meg a **true** és a **false** implementációját a **/bin** vagy **/usr/bin** jegyzékben! (Hogyan találhatjuk ki, hogy melyikben vannak?) □

5-10. feladat. Módosítsuk a **watchfor** parancsot úgy, hogy a többszörös argumentumot több emberként kezelje, ne kelljen *'joe|mary'*-t írnunk! □

5-11. feladat. Írjunk egy **watchwho** változatot, amely az **awk** helyett a **comm**-ot használja a régi és az új adat összehasonlítására! Melyik változat tetszik jobban? □

5-12. feladat. Írjunk olyan **watchwho** változatot, amely a **who** kimenetét fájlok helyett burokváltozókban tárolja! Melyik változat tetszik jobban? Melyik változat gyorsabb? Kell-e a **watchwho**-nak és a **checkmail**-nak automatikus &-et csinálnia? □

5-13. feladat. Mi a különbség a burok kettőspont (**:**, **semmit_se_csinálj**) parancsa és # megjegyzéskaraktere között? Szükség van mindkettőre? □

Burokváltozók értékadása

$var	var értéke; semmi, ha var nincs definiálva
${var}	ugyanaz; hasznos, ha alfanumerikus karakterek követik a változó nevét
${var-dolog}	var értéke, ha definiálva van, egyébként dolog. $var változatlan
${var=dolog}	var értéke, ha definiálva van, egyébként dolog. Ha definiálatlan, akkor $var értéke dolog lesz
${var?üzenet}	ha definiált, akkor $var; ha nem, akkor kiírja az üzenet-et és a burok kilép. Ha az üzenet üres, akkor ezt írja: var: parameter
${var+dolog}	dolog, ha $var definiálva van, egyébként semmi

5.4. Csapdák: megszakítások elfogása

Ha a **watchwho** futása közben megnyomjuk a *DEL* gombot, vagy megszakítjuk a vonalat (lerakjuk a telefonkagylót), akkor egy vagy két ideiglenes fájl van a **/tmp**-ben. A **watchwho** parancsnak kilépés előtt el kell tüntetnie az ideiglenes fájlokat. Valahogyan érzékelnünk kell, hogy ilyen esemény történik, és tudnunk kell helyrehozni a dolgokat.

Ha megnyomjuk a *DEL* gombot, akkor a rendszer egy *megszakításjelet* küld az azon a terminálon futó minden folyamathoz. Hasonlóan, ha megszakítjuk a vonalat, akkor a rendszer egy *vonalszakadás jelet* küld. (Vannak más jelek is.) Ha a programnak nem kell a jel hatására valamilyen explicit feladatot végeznie, akkor a jel érkezésekor a program befejeződik. A burok az *&*-del futó programokat védi a megszakítástól (de a vonalszakadástól nem).

A 7. fejezet részletesen foglalkozik a jelekkel, de nem kell túl sokat tudnunk róluk ahhoz, hogy a burokban kezelni tudjuk őket. A burok beépített **trap** parancsában megadhatjuk a jel észlelésekor végrehajtandó parancsok sorozatát:

trap *parancs_sorozat jelszám_lista*

A *parancs_sorozat* egyetlen argumentum, ezért majdnem mindig fél idézőjelek közé kell tenni. A *jelszámok* az egyes jeleket azonosító kis egész számok. (Pl. a 2 a *DEL* megnyomásakor, az 1 a telefonvonal megszakításakor keletkezik.) A burokprogramozáskor leggyakrabban használatos jelszámokat az 5.4. táblázat tartalmazza.

Így ahhoz, hogy a **watchwho** az ideiglenes fájlokat eltüntesse, a ciklus előtt egy **trap**-hívásnak kell lennie, amely elcsípi a vonalszakadást, a megszakítást és a befejeződést:

Burokjelszámok

0	burokkilépés (bármilyen okból, beleértve a fájl végét is)
1	vonalszakadás
2	megszakítás (*DEL* billentyű)
3	kilépés (*ctl-* ; a program tártartalom-kiírást csinál)
9	leállítás (nem lehet elfogni vagy törölni)
15	befejeződés, a **kill(1)** alapértelmezés szerint előállított jele

```
...
trap 'rm -f $new $old; exit 1' 1 2 15

while :
...
```

A **trap** első argumentumát adó parancssorozat olyan, mint egy szubrutin, amely a jel érzékelésekor azonnal meghívódik. Amikor befejeződik az első argumentumban adott program futása, akkor a program a megszakítás helyénél folytatódik, hacsak a jel nem állította le azt. Ezért a **trap** parancssorozatban explicite meg kell hívni az **exit**-et, különben a burokprogram a megszakítás után folytatni fogja a program végrehajtását. Továbbá a parancssorozatot a burok kétszer fogja beolvasni: először a **trap** beállításakor, másodszor meghívásakor. Ezért a parancssorozatot a legjobb fél idézőjellel védeni, mert így a változók csak a **trap** rutinok végrehajtásakor kapnak értéket. Ebben az esetben ugyan nincs különbség a kettő között de később fogunk látni olyan esetet, amikor lesz. (Egyébként a **-f** opció az **rm** számára azt jelenti, hogy ne tegyen fel kérdéseket.)

A **trap** interaktív alkalmazásokban is hasznos lehet. Gyakran így védjük meg a programot, nehogy a telefonvonal hibája miatt létrejött vonalszakadás leállítsa:

```
$ (trap '' 1; hosszanfutó_parancs) &
2134
$
```

A **null** parancssorozat ('') azt jelenti, hogy ebben a folyamatban és gyermekeiben ne vegye figyelembe a megszakításokat. A zárójel hatására a **trap** és a parancs egy háttér-alburokban együtt fut, enélkül a **trap** a bejelentkezési burokra és a *hosszanfutó_parancs*-ra egyaránt vonatkozna.

A **nohup(1)** parancs egy rövid burokprogram, amely ellátja ezt a feladatot. Íme a 7. kiadásbeli változat a maga teljességében:

```
$ cat `which nohup`
trap "" 1 15
if test -t 2>&1
then
        echo "Kimenet küldése 'nohup.out'-ra"
        exec nice -5 $* >>nohup.out 2>&1
else
        exec nice -5 $* 2>&1
fi
$
```

A **test -t** megnézi, hogy a szabványos kimenet egy terminál-e, ebből tudja, hogy a kimenetet ki kell-e menteni. A háttérprogram a **nice**-szal fut, hogy az interaktív programokénál alacsonyabb prioritása legyen. (Figyeljük meg, hogy a **nohup** nem állítja be a PATH-ot! Kellene?)

Az **exec** csak a hatékonyság kedvéért van, a parancs anélkül is ugyanúgy futna. Az **exec** a burok egy beépített utasítása, amely az ezt a burkot futtató folyamatot összekapcsolja a megnevezett programmal, ezáltal egy folyamatot megtakarít – azt a burkot, amely normál esetben megvárja a program befejeződését. Sok más helyen is használhattuk volna az **exec**-et, mint pl. a továbbfejlesztett **cal** program végén a **/usr/bin/cal** hívásakor.

Egyébként a 9-es jelet nem lehet elfogni vagy figyelmen kívül hagyni, az mindig leállít. A burokból ezt a következő formában küldhetjük ki:

```
$ kill -9 folyamatazonosító ...
```

A **kill -9** nem alapértelmezés, mert az így leállított folyamatnak nincs esélye, hogy halála előtt rendbeszedje a dolgait.

5-14. feladat. A **nohup** fenti változatában a parancs szabványos hibájának kimenete és a szabványos kimenet megegyezik. Jó tervezés ez? Ha nem, hogyan választanánk szét azokat egyértelműen? □

5-15. feladat. Keressük meg a burok beépített **times** utasítását és írjunk be egy sort **.profile**-unkba úgy, hogy kijelentkezéskor a burok kiírja, mennyi CPU időt használtunk fel! □

5-16. feladat. Írjunk olyan programot, amely a **/etc/passwd**-ben megkeresi a következő elérhető felhasználói azonosítót! Ha van elég lelkesedésünk (és engedélyünk), alakítsuk ezt olyan paranccsá, amely a rendszerhez egy új felhasználót tesz hozzá! Milyen engedély kell ehhez? Hogyan kell a megszakításokat kezelnie? □

5.5. Fájlcsere: overwrite

A **sort** parancs **−o** opciója felülírja a fájlt:

```
$ sort file1 −o file2
```

ugyanaz, mint

```
$ sort file1 >file2
```

Ha a **file1** és **file2** ugyanaz a fájl, akkor a > átirányítás a rendezés előtt csonkítja a bemeneti fájlt. A **−o** opcióval viszont jól működik, mert a bemenetet rendezi és egy ideiglenes fájlon megőrzi, mielőtt létrehozná a kimeneti fájlt.

Sok egyéb parancs használhatná a **−o** opciót. Pl. a **sed** helyben szerkeszthetne egy fájlt:

```
$ sed 's/UNIX/UNIX(TM)/g' sz2 −o sz2   Így nem működik!
```

Nem lenne praktikus, hogy minden ilyen parancsot módosítsunk csupán azért, hogy hozzátegyük ezt az opciót. Ráadásul ez még rossz tervezés is lenne: célszerűbb centralizálni a tevékenységeket, ahogy azt a burok a > operátorral teszi. Készítsünk egy **overwrite** nevű, ezt a feladatot végrehajtó programot! Az első terv lehet pl.:

```
$ sed 's/UNIX/UNIX(TM)g' sz2 | overwrite sz2
```

A megvalósítás világos − a bemenetet a fájl végéig megőrizzük, ezután az adatot az argumentumfájlba másoljuk:

```
# overwrite: a szabványos bemenetet fájl vége után a kimenetre másolja
# 1. változat, még hibás
PATH=/bin:/usr/bin
case $# in
1)      ;;
*)      echo 'Usage: overwrite file' 1>&2; exit 2
esac
new=/tmp/overwr.$$
trap 'rm −f $new; exit 1' 1 2 15
cat >$new       # összegyűjti a bemenetet
cp $new $1      # felülírja a bemeneti fájlt
rm −f $new
```

Azért használtunk **mv** helyett **cp**-t, hogy a kimeneti fájl engedélyei és tulajdonosa, ha már léteznek, ne változzanak meg.

Ennek az egyszerűnek látszó változatnak van egy végzetes hibája: ha a felhasználó a **cp** során megnyomja a *DEL* billentyűt, akkor az eredeti bemeneti fájl tönkremegy. Meg kell akadályoznunk tehát, hogy a bemeneti fájl felülírását egy megszakítás megállítsa:

```
# overwrite: a szabvány bemenetet fájl vége után a kimenetre másolja
# 2. változat, még ez is hibás

PATH=/bin:/usr/bin
case $# in
1)      ;;
*)      echo 'Usage: overwrite file' 1>&2; exit 2
esac

new=/tmp/overwr1.$$
old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15
cat >$new           # összegyűjti a bemenetet
cp $1 $old          # megőrzi az eredeti fájlt
trap '' 1 2 15      # megzavartak, ne törődjünk a jelekkel
cp $new $1          # felülírjuk a bemeneti fájlt
rm -f $new $old
```

Ha az eredeti fájl érintése előtt *DEL* érkezik, akkor az ideiglenes fájlok törlődnek, az eredeti fájl pedig változatlanul megmarad. Ha a háttért már létrehozta, akkor a jelek figyelmen kívül maradnak, nehogy az utolsó **cp**-t valami megszakítsa – ha a **cp** már elindult, akkor az **overwrite**-nek kötelessége az eredeti fájl megváltoztatása.

Még mindig van egy kényes kérdés. Tekintsük a következőt:

```
$ sed 's/UNIX/UNIX(TM)g' pontos | overwrite pontos
command garbled: s/UNIX/UNIX(TM)g
$ ls -l pontos
-rw-rw-rw-  1 you   0 Oct 1 09 : 02 pontos   #$%@*!
$
```

Ha az **overwrite** bemenetét adó program hibás, akkor annak kimenete üres lesz és az **overwrite** kötelességtudóan és megbízhatóan tönkreteszi az argumentumfájlt.

A hibalehetőség elkerülésére több megoldás van. Az **overwrite** a fájl áthelyezése előtt kérhet megerősítést, de ha interaktívvá tesszük az **overwrite**-ot, akkor sokat veszít az érdemeiből. Az **overwrite** ellenőrizheti (a **test** **-z**-vel) hogy nem üres-e a bemenet (de ez csúnya és nem helyes), vagy pedig előállíthat valamilyen kimenetet, mielőtt hibát észlel. A legjobb megoldás, ha az adatgeneráló programot az **overwrite** vezérlése alatt futtatjuk, ezzel ellenőrizhetjük annak kilépési státusát. Ez a hagyomány és az intuíció ellen van – egy csőhálózatban az **overwrite** normál esetben az utolsó. Mégis ahhoz, hogy jól működjön, az első helyre kell kerülnie. Az **overwrite** saját szabványos kimenetére semmit sem ír, így a dolog nem megy az általánosság rovására. Szintaxisa sem egyedülálló: a **time**, **nice** és **nohup** mind olyan parancsok, amelyek argumentumként egy másik parancsot használnak.

Íme a biztonságos változat:

```
# overwrite: a szabványos bemenetet fájl vége után a kimenetre másolja
# végleges változat
```

```
opath=$PATH
PATH=/bin:/usr/bin

case $# in
0 | 1)    echo 'Usage: overwrite file cmd [args]' 1>&2; exit 2
esac

file=$1; shift
new=/tmp/overwr1.$$; old=/tmp/overwr2.$$
trap 'rm -f $new $old; exit 1' 1 2 15  # kiüríti a fájlokat

if PATH=$opath "$@" >$new    # összegyűjti a bemenetet
then
  cp $file $old       # megőrzi az eredeti fájlt
  trap '' 1 2 15      # megzavartak, ne törődjünk a jelekkel
  cp $new $file
else
  echo "overwrite: $1 failed, $file unchanged" 1>&2
  exit 1
fi
rm -f $new $old
```

A burok beépített **shift** parancsa az argumentumlistát egy pozícióval balra mozdítja: *$2*-ből *$1*, *$3*-ból *$2* lesz és így tovább. *$@* az összes (**shift** utáni) argumentumot adja, ugyanúgy, mint a *$**, de nem értelmezi azokat. Erre az 5.7. szakaszban még visszatérünk.

Figyeljük meg, hogy a **PATH**-ot a felhasználói parancsok futtatásához helyreállítottuk; ha nem tettük volna, akkor azok a parancsok, amelyek nem a **/bin** vagy **/usr/bin** alatt vannak, az **overwrite** számára elérhetetlenek lennének.

Az **overwrite** most már működik (bár kicsit esetlenül):

```
$ cat notice
UNIX is a Trademark of Bell Laboratories
$ overwrite notice sed 's/UNIXUNIX(TM)/g' notice
command garbled: s/UNIXUNIX(TM)/g
overwrite: sed failed, notice unchanged
$ cat notice
UNIX is a Trademark of Bell Laboratories          Változatlan!
$ overwrite notice sed 's/UNIX/UNIX(TM)/g' notice
$ cat notice
UNIX(TM) is a Trademark of Bell Laboratories
$
```

A **sed**-et gyakran használjuk arra, hogy egy szó minden előfordulását egy másikra cseréljünk. Ha kéznél van az **overwrite**, akkor egy burokfájllal egyszerűen automatizálhatjuk a feladatot:

```
$ cat replace
# replace: a fájlokban str1-et str2-re cseréli
PATH=/bin:/usr/bin
```

```
case $# in
0|1|2)    echo 'Usage: replace str1 str2 files' 1>&2; exit 1
esac

left="$1"; right="$2"; shift; shift

for i
do
    overwrite $i sed "s@$left@$right@g" $i
done
$ cat lábjegyzet
UNIX nem betűszó
$ replace UNIX Unix lábjegyzet
$ cat lábjegyzet
Unix nem betűszó
$
```

(Emlékezzünk vissza, hogy ha a **for** utasítás listája üres, akkor az alapértelmezés *$**.) A helyettesítési parancsban / helyett @ karaktert használtunk, mert a @ kisebb valószínűséggel okoz a bemenő karakterláncban zavart.

A **replace** a **PATH**-ot **/bin:/usr/bin**-re állítja, a **$HOME/bin** kirekesztésével. Ez azt jelenti, hogy a **replace** működéséhez az **overwrite**-nak az **/usr/bin** alatt kell lennie. Ezt csupán az egyszerűség kedvéért feltételeztük; ha nem tudjuk az **overwrite**-ot a **/usr/bin**-be beépíteni, akkor a **replace**-ben a **PATH**-ba betesszük a **$HOME/bin**-t, vagy explicite megadjuk az overwrite ösvénynevét. A továbbiakban feltételezzük és úgy értjük, hogy az általunk írt parancsok a **/usr/bin**-ben vannak.

5-17. feladat. Miért nem használja az **overwrite** a **trap**-ben a 0 jelet, hogy kilépéskor törölje a fájlokat? Útmutatás: próbáljunk *DEL*-t beütni az alábbi program futása közben:

```
trap "echo exiting; exit 1" 0 2
sleep 10
```
□

5-18. feladat. Írjunk a **replace**-hez olyan **−v** opciót, hogy kiírja a **/dev/tty** megváltozott sorait! Fontos útmutatás:
`s/$left/$right/g/$vflag` □

5-19. feladat. Javítsuk ki úgy a **replace**-t, hogy a helyettesítő karakterláncban levő karakterektől függetlenül működjön! □

5-20. feladat. Lehet-e a **replace**-t arra használni, hogy a programban az **i** változót mindenhol **index**-re cserélje? Hogyan kell megváltoztatni, hogy így működjön? □

5-21. feladat. Kényelmes és hatékony dolog-e, hogy a **replace** a **/usr/bin**-hez tartozik? Nem volna előnyösebb egyszerűen a megfelelő **sed** parancsokat begépelni? Miért? Vagy miért nem? □

5.22. feladat (nehéz).

```
$ overwrite file 'who | sort'
```

nem működik. Magyarázzuk meg, miért nem, és javítsuk ki! Útmutatás: nézzük meg a sh(1)-ben az **eval**-t! Hogyan hat a megoldásunk a parancs metakaraktereinek értelmezésére? □

5.6. Folyamatok név szerinti leállítása: zap

A **kill** parancs csak a folyamatazonosítóval megadott folyamatokat állítja le. Ha egy háttérfolyamatot kell leállítanunk, akkor általában le kell futtatnunk a **ps** programot ahhoz, hogy megtudjuk a folyamatazonosítót. Ezután a **kill** argumentumaként újra be kell gépelnünk ugyanazt az azonosítót. Milyen butaság, hogy az egyik program kiír egy számot, amit manuálisan rögtön visszaírunk a másiknak. Miért ne készítsünk egy, mondjuk a **zap** nevű programot, amely automatikusan elvégzi a feladatot?

Az egyik magyarázat az, hogy a folyamatok leállítása veszélyes, nagyon gondosan kell eljárnunk, hogy a megfelelő folyamatot töröljük. Védelmet jelent, ha a **zap**-ot mindig interaktívan futtatjuk, és a **pick**-kel kerestetjük ki az áldozatokat.

Rövid **pick** összefoglaló: sorra kiírja az argumentumait és választ kér a felhasználótól. Ha a válasz *y*, akkor az argumentumot kiírja (a **pick** a következő szakasz anyaga). A **zap** a **pick**-et annak ellenőrzésére használja, hogy a név szerint kiválasztott folyamatok valóban azok-e, amelyeket a felhasználó törölni kíván?

```
$ cat zap
# zap minta: leállítja a mintával egyezõ folyamatokat
# HIBÁS változat

PATH=/bin:/usr/bin

case $# in
0)      echo 'Usage: zap minta' 1>&2; exit 1
esac

kill `pick \ `ps -ag | grep "$*"\ `| awk '{print $1}'`
$
```

Figyeljük meg az egymásba ágyazott fordított törtvonalakkal védett fordított fél idézőjeleket! Az **awk** program a **pick** által kiválogatott **ps** kimenetből kiválasztja a folyamatazonosítót.

```
$ sleep 1000 &
22126
$ ps -ag
   PID  TTY  TIME  CMD
```

```
...
 22126  0      0 : 00  sleep 1000
...
$ zap sleep
22126?
0? q                                  Mi történik itt?
$
```

Az a baj, hogy a **ps** kimenetét szavakra tördeltük. Ezeket a **pick** külön argumentumnak látja, ahelyett, hogy az egész sort egyszerre dolgozná fel. A burok normál viselkedése az, hogy a karakterláncokat szóköz/nem szóköz határoknál tördeli argumentumokká, mint pl. itt:

```
for i in 1 2 3 4 5
```

Ebben a programban a burok működését úgy kell vezérelnünk, hogy amikor a karakterláncokat argumentumokká bontja, akkor csak az újsor-karakterek válasszák szét a szomszédos „szókat".

Az **IFS** burokváltozó (internal field separator: belső mezőelválasztó) azon karakterek lánca, amelyek a szavakat az argumentumlistákban (fordított fél idézőjel, **for** utasítás) elválasztják. Általában az **IFS** tartalma szóköz, tabulátor és újsorkarakter, de ezt szükség szerint megváltoztathatjuk, pl. csak újsorra:

```
$ echo 'echo $#' >nargs
$ cx nargs
$ who
you    tty0   Oct 1 05 : 59
pjw    tty2   Oct 1 11 : 26
$ nargs `who`
10                    Tíz mező szóköz- és újsor-elválasztóval.
$ IFS='
'                     Csak újsor az elválasztó.
$ nargs `who`
2                     Két sor, két mező.
$
```

Ha az **IFS**-t újsorra állítjuk be, akkor a **zap** jól működik:

```
$ cat zap
# zap minta: leállítja a mintával egyező folyamatokat
# utolsó változat

PATH=/bin:/usr/bin
IFS='
'                     # csak egy újsor az elválasztó
case $1 in
"")     echo 'Usage: zap [-2] pattern' 1>&2; exit 1 ;;
-*)     SIG=$1; shift
esac
```

```
echo ' PID TTY   TIME CMD'
kill $SIG `pick \`ps -ag | egrep "$*"\` | awk '{print $1}'`
$ ps -ag
  PID  TTY  TIME  CMD
...
22126  0    0:00  sleep 1000
...
$ zap sleep
  PID  TTY  TIME  CMD
22126  0    0:00  sleep 1000? y
23104  0    0:02  egrep sleep? n
$
```

Néhány ötletettel még kiegészítettük programunkat hozzátettünk egy opcionális argumentumot a jel megadására (megjegyezzük, hogy ha nem adunk argumentumot, akkor a **SIG** definiálatlan lesz és üres karakterláncként kezeli a program), a **grep** helyett pedig az **egrep**-et használtuk, mert így bonyolultabb minták, mint pl. a *'sleep|date'* is megengedhetők. A kezdő **echo** írja a **ps** kimenete elé az oszlopfejlécet.

Miért hívjuk ezt a parancsot **zap**-nak ahelyett, hogy egyszerűen **kill**-nek nevezzénk? Ennek fő oka az, hogy ellentétben **cal** példánkkal, valójában nem alkottunk új **kill** parancsot; a **zap** egy feladatra jól használható interaktív program – mellette a **kill**-t eredeti formájában is meg akarjuk tartani. Továbbá a **zap** bosszantóan lassú, az extra programok gépideje tekintélyes, bár a **ps** (aminek mindenképpen futnia kell) a legköltségesebb. A következő fejezetben egy sokkal hatásosabb változatot mutatunk be.

5-23. feladat. Módosítsuk a **zap**-ot úgy, hogy a **ps** a fejlécet a csőhálózatból úgy írja ki, hogy az érzéketlen legyen a **ps** kimenetformátumának változásaira! Mennyire bonyolítja ez a programot? □

5.7. Szóközök és argumentumok kezelése: pick

Mindent összegyűjtöttünk ahhoz, hogy a burokba egy **pick** parancsot írjunk. Már csak egy olyan mechanizmus kell, amely képes a felhasználói bemenetet olvasni. A burok beépített **read** parancsa a szabványos bemenetről beolvas egy szövegsort, és a szöveget (újsor nélkül) értékként a megadott változókhoz rendeli:

```
$ read üdvözlet
jó reggelt                Új értéket írunk az üdvözlet-be.
$ echo $üdvözlet
jó reggelt
$
```

A **read** leggyakoribb használata: a környezet beállítása a **.profile**-ban bejelentkezéskor; értékadás az olyan burokváltozóknak, mint pl. a **TERM**.

A **read** csak a szabványos bemenetről tud olvasni; nem lehet átirányítani. A burok beépített parancsait (ellentétben az olyan egyszerű vezérlőelemekkel, mint pl. a **for**) nem lehet a > vagy < jellel átirányítani.

```
$ read üdvözlet </etc/passwd
viszontlátásra          Mindenképpen be kell írni egy értéket.
illegal io              A burok most hibát jelez.
$ echo $üdvözlet
viszontlátásra          Az üdvözletnek a beírt értéke van, nem a fájlból való.
$
```

Ezt a burok hibájának tekinthetjük, de hát ilyen az élet! Szerencsére ez rendszerint kijátszható azzal, hogy a **read**-et körülvevő ciklust irányítjuk át. Ez a mi **pick** parancsmegvalósításunk kulcsa is:

```
# pick: argumentumok válogatása.

PATH=/bin:/usr/bin

for i                   # minden argumentumra
do
        echo -n "$i?" >/dev/tty
        read válasz
        case $válasz in
        y*) echo $i ; ;
        q*) break
        esac
done </dev/tty
```

Az **echo** −n elnyomja a záró újsort, így a választ a prompt sorába írhatjuk. Továbbá a promptokat természetesen kiírja a **/dev/tty**-re, hiszen majdnem biztos, hogy a szabványos kimenet nem a terminál.

A C nyelvből kölcsönzött **break** utasítás megszakítja a legbelső ciklust. Esetünkben *q* beírása kiléptet a **for** ciklusból. Azért választottuk lezárónak a *q*-t, mert egyszerű, kényelmes és más programokkal egyező.

Érdekes a **pick** argumentumaiban levő szóközökkel játszani:

```
$ pick '1 2' 3
1 2?
3?
$
```

Ha látni akarjuk, hogyan olvassa a **pick** az argumentumait, futtassuk, és minden **$** után nyomjunk *RETURN*-t. Pontosan úgy működik ahogy vártuk, a **for i** helyesen kezeli az argumentumokat. Másként is írhattuk volna a ciklust:

```
$ grep for pick         Lássuk, mit tud ez a változat!
for i in $*
```

```
$ pick '1 2' 3
1?
2?
3?
$
```

Ebben a formában nem működik, mert a ciklus operandusait újravizsgálja, és a szóközök miatt az első argumentumból két argumentum lesz. Próbáljuk idézőjelbe tenni az *$**-ot:

```
$ grep for pick          Másik változatot próbálunk.
for i in "$*"
$ pick '1 2' 3
1 2 3?
$
```

Ez sem működik, mert *"$*"* a szóközökkel elválasztott összes argumentumból egyetlen szót alkot. Természetesen van megoldás, de az tiszta bűvészkedés: a *"$@"* karakterláncot a burok különlegesen kezeli, és pontosan az argumentumokat adja át a burokfájlnak:

```
$ grep for pick          Harmadszor próbálkozunk.
for i in "$@"
$ pick '1 2' 3
1 2?
3?
$
```

Ha a *$@* nincs idézőjelek között, akkor ugyanaz, mint *$**; csak idézőjelek között viselkedik különlegesen! Az **overwrite**-ban a felhasználói parancs argumentumának megőrzésére már használtuk ezt.

Összefoglalva a szabályok a következők:

- *$** és *$@* az argumentumokba nyúlnak, azokat újból végignézik, az argumentumokban levő szóközök többszörös argumentumokra vezetnek.
- *"$*"* a burokfájl számára egyetlen szó, amely az összes argumentumból szóközökkel van összekapcsolva.
- A *"$@"* azonos a burokfájl által kapott argumentumokkal; az argumentumok szóközét elhagyja, az eredmény az eredeti argumentumokkal megegyező szólista.

Ha a **pick**-nek nincs argumentuma, akkor valószínűleg a szabványos bemenetét fogja olvasni, így

```
$ pick `cat postalista`
```

helyett ezt mondhatjuk:

```
$ pick <postalista
```

Ne mélyedjünk el a **pick**-nek ebben a változatában, mert bonyodalmakhoz jutunk, és az egész sokkal bonyolultabb, mint a C nyelvben írt ugyanilyen program, amelyet a következő fejezetben ismertetünk.

Az alábbi feladatok közül az első kettő bonyolult, de a haladóbb burok programozók számára tanulságos.

5-24. feladat. Próbáljunk írni egy olyan **pick**-et, amely argumentumait a szabványos bemenetről olvassa, ha a parancssorban nincsenek argumentumok! A szóközöket helyesen kell kezelni. Működik a *q* válasz? Ha nem, próbálkozzunk a következő feladattal! □

5-25. feladat. Bár a **read** és **set**, mint a burok beépített utasításai nem átirányíthatók, magát a burkot ideiglenesen át lehet irányítani. Olvassuk el az sh(1) leírás **exec** parancsot ismertető részét, és dolgozzuk ki, hogyan olvasson a **read** a **/dev/tty**-ből alburok hívás nélkül! (Ha nem megy, előbb olvassuk el a 7. fejezetet!) □

5-26. feladat (sokkal könnyebb). Használjuk a **.profile**-unkban a **read**-et a **TERM** és minden ettől függő érték, mint pl. a tabulátorstopok beállítására! □

5.8. Közhasznú üzenetek: news

Az első fejezetben már említettük, hogy Olvasóink rendszerében lehet egy **news** parancs, amely a felhasználók általános érdeklődésére számot tartó üzeneteket közvetít. Bár a parancs neve és a részletek különbözőek lehetnek, majdnem minden rendszerben van újságszolgálat. Saját **news** parancsunkat nem azért ismertetjük, hogy Olvasóink saját rendszerükét erre kicseréljék, hanem hogy bemutassuk, milyen egyszerűen lehet a burokban egy ilyen programot írni. Érdekes lenne a mi **news** parancsunkat összehasonlítani Olvasóink rendszerének változatával.

Az ilyen programok alapgondolata rendszerint az, hogy az egyes újságcikkeket egy külön jegyzékben, mondjuk **/usr/news** alatt tárolják, minden cikket külön fájlban. A **news** (vagyis a mi **news** programunk) úgy működik, hogy a **/usr/news** alatti fájlok módosítási idejét összehasonlítja a saját jegyzékünkben levő, postabélyegzőként szolgáló **.news_time** fájllal. Programbelövéshez használhatjuk a **.** jegyzéket mind az újságfájlok, mind a **.news_time** fájl jegyzékeként. Ha a program általános használatra kész, akkor ezt meg lehet változtatni **/usr/news**-ra.

```
$ cat news
# news: újságfájlok kiíratása, 1. változat

HOME=.          # csak belövéshez
cd .            # /usr/news helyett
for i in `ls -t * $HOME/.news_time`
do
        case $i in
```

```
        */.news-time)           break ; ;
        *)                      echo news: $i
        esac
done
touch $HOME/.news_time
$ touch .news_time
$ touch x
$ touch y
$ news
news: y
news: x
$
```

A **touch** argumentumfájljának utolsó módosítási idejét az aktuális időre változtatja anélkül, hogy magát a fájlt módosítaná. A programbelövéshez csak visszhangoztatjuk az újságfájlok neveit, nem íratjuk ki azokat.

A ciklus akkor fejeződik be, ha megtalálja a **.news_time** fájlt, ezért csak az újabb fájlokat listázza. Megjegyezzük, hogy a **case** utasításban a * a / jellel is egyezést mutathat, ami pedig fájlnevek mintáiban kellemetlen.

Mit csinál a program, ha a **.news_time** nem létezik?

```
$ rm .news_time
$ news
$
```

Váratlanul és hibásan hallgat. Ugyanis akkor, ha az **ls** nem találja a fájlt, jelzi ezt a szabványos kimenetén, még mielőtt a létező fájlokról bármilyen információt kiírna. Letagadhatatlan hiba – a diagnosztikát ki kell írni a szabványos hiba kimenetére. Elkerülhetjük ezt, ha a ciklusban felismerjük a hibát és a szabványos hibát átirányítjuk a szabványos kimenetre. Ezzel elérhetjük, hogy minden változat egyformán működjön. (Ezt a kellemetlenséget a rendszer újabb változataiban már kiszűrték, de itt meghagytuk úgy, ahogy volt. Megmutatjuk, hogyan győzedelmeskedhetünk a kisebb hibák felett.)

```
$ cat news
# news: újságfájlok kiíratása, 2. változat

HOME=.          # csak belövéshez
cd .            # /usr/news helyett
IFS='
'               # csak újsor
for i in `ls -t * $HOME/.news_time 2>&1`
do
        case $i in
        *' not found') ; ;
        */.news_time)           break ; ;
        *)                      echo news: $i ; ;
        esac
done
```

```
touch $HOME/.news_time
$ rm .news_time
$ news
news: news
news: y
news: x
$
```

Az **IFS**-t újsorra kellett állítanunk, nehogy a

```
./.news_time not found
```

üzenet három szóként értelmeződjön.

Most az következik, hogy a **news** írja is ki a fájlokat, ne csak a nevüket visszhangozza. Hasznos tudnunk, hogy ki és mikor küldte az üzenetet, ezért a **set** és az **ls −l** parancsok felhasználásával fejlécet íratunk az üzenet elé:

```
$ ls −l news
−rwxrwxrwx 1 you   208 Oct 1 12 : 05 news
$ set `ls −l news`
−rwxrwxrwx: bad option(s)        Valami hibás!
$
```

Íme egy példa, amikor a burokban bajt okoz a program és az adat felcserélhetősége. A **set** azért panaszkodik, mert argumentuma (−rwxrwxrwx) mínuszjellel kezdődik, és ezért olyan, mintha opció lenne. Egyszerű (bár nem elegáns) megoldás, ha az argumentum elé egy közönséges karaktert írunk:

```
$ set X`ls −l news`
$ echo "news: ($3) $5 $6 $7"
news :(you) Oct 1 12 : 05
$
```

Ez elfogadható formátum, a fájlnévvel együtt mutatja az üzenet szerzőjét és a dátumot.

Íme a **news** parancs végső változata:

```
# news: újságfájlok kiíratása, végleges változat

PATH=/bin:/usr/bin
IFS='
'                    # csak újsor az elválasztó
cd /usr/news

for i in `ls −t * $HOME/.news_time 2>&1`

do
```

```
        IFS=' '
        case $i in
        *' not found') ; ;
        */.news_time) break ; ;
        *)      set X`ls -l $i`
                echo "
$i: ($3) $5 $6 $7
"
                cat $i
        esac
done
touch $HOME/.news_time
```

A fejléc új sorai kiíratáskor elválasztják az újság cikkeit. Az **IFS** első értéke csak egy újsor-karakter, így az első **ls**-ből származó not found üzenet (ha van ilyen) egyetlen argumentum. A második értékadással **IFS** szóköz értéket kap, így a második **ls** kimenete több argumentumra bomlik.

5-27. feladat. Bővítsük a **news** parancsot egy **-n** (notify) opcióval, amely jelzi, de nem írja ki az újságcikkeket és a **.news_time**-ot nem érinti! Ezt elhelyezhetjük **.profile** fájlunkban. □

5-28. feladat. Hasonlítsuk össze az itt leírt **news**-tervezést és -megvalósítást az Olvasóink rendszerén levő hasonló paranccsal! □

5.9. Fájlváltozások követése: get és put

Hosszúra nyúlt fejezetünknek ebben az utolsó szakaszában egy nagyobb, bonyolultabb példával mutatjuk be, hogyan működhet együtt a burok az **awk** és **sed** programokkal.

A programok fejlesztése során a talált hibákat kijavítjuk, a programot újabb és újabb tulajdonságokkal bővítjük. Néha kényelmes és hasznos a különböző változatok megőrzése, különösen, ha más gépekre is átviszik a programokat; gyakran jönnek ugyanis a kérdések: Mit változtattatok azóta, hogy a miénket megkaptuk?" vagy „Hogyan javítottátok ki az ilyen-meg-ilyen hibát?". A háttérmásolatok biztonságosabbá teszik az új ötletek kipróbálását is; ha valami nem jó sül el, fájdalommentesen vissza lehet térni az eredeti programhoz.

Az összes változat tárolása lehet az egyik megoldás, ezt azonban bonyolult megszervezni és sok lemezterületet igényel. Ehelyett kihasználjuk, hogy az egymás utáni változatok rendszerint sok azonos részt tartalmaznak, amelyeket elég csak egyszer tárolnunk. A **diff -e** parancs előállítja azoknak az **ed** parancsoknak a listáját, amelyek az **old** fájlt **new** fájllá alakítják:

```
$ diff -e old new
```

Így egy fájl összes változatát egyetlen (különböző) fájlban tárolhatjuk oly módon, hogy egy teljes változatot tárolunk együtt az ezt a változatot bármely másik változattá alakító szerkesztőparancsokkal.

Nyilvánvalóan két szervezési lehetőség van: a legújabb változatot érintetlenül hagyjuk, és az időrendben visszafelé szerkesztő parancsokat gyűjtjük egybe, vagy a legrégibb változatot tároljuk és abból az előreszerkesztő parancsok segítségével kapjuk az újabbakat. Bár ez utóbbi programozása valamivel egyszerűbb, az előbbi gyorsabb, különösen ha sok változat van, hiszen általában a legutolsóra vagyunk kíváncsiak.

Mi az első szervezési lehetőséget választottuk. Egyetlen fájlban, amit *történetfájlnak* fogunk nevezni, lesz az aktuális változat, amit azok a szerkesztőparancs-sorozatok követnek, amelyek az egyes változatokat az előzővé (vagyis a következő régebbivé) alakítják. Minden szerkesztőparancs-sorozat ilyen sorral kezdődik:

```
@@@ személy dátum összefoglalás
```

Az **összefoglalás** (summary) a változásoknak a **személy** (a változtató) által adott egysoros leírása.

A változatok karbantartására két parancsunk van: a **get** elővesz a történetfájlból egy változatot, a **put** az egysoros változás-összefoglalás megadása után új változatot léptet be a történetfájlba.

A megvalósítás bemutatása előtt lássunk egy példát, hogyan működik a **get**, a **put** és a történetfájl karbantartása:

```
$ echo egy sor szöveg >vacak
$ put vacak
Summary: új fájl létrehozása          Beírjuk a leírást.
get: no file vacak.T                  Történetfájl nem létezik...
put: creating vacak.T                 ...így a put létrehozza azt.
$ cat vacak.T
egy sor szöveg
@@@ you Sat Oct 1 13 : 31 : 03 EDT 1983 új fájl létrehozása
$ echo újabb sor >>vacak
$ put vacak
Summary: egy sor hozzáírása
$ cat vacak.T
egy sor szöveg
újabb sor
@@@ you Sat Oct 1 13 : 32 : 28 EDT 1983 egy sor hozzáírása
2d
@@@ you Sat Oct 1 13 : 31 : 03 EDT 1983 új fájl létrehozása
$
```

A szerkesztőparancsok jelen esetben az egyetlen **2d** tartalmú sorból állnak. Ez a fájl második sorát törölve az új változatot az eredetivé alakítja.

```
$ rm vacak
$ get vacak                           A legutolsó változat.
```

```
$ cat vacak
egy sor szöveg
újabb sor
$ get -1 vacak
$ cat vacak                          Az eggyel régebbi változat.
egy sor szöveg
$ get vacak                          Újra a legutolsó.
$ replace újabb másik vacak          Megváltoztatjuk.
$ put vacak
Summary: második sor változtatása
$ cat vacak.T
egy sor szöveg
másik sor
@@@ you Sat Oct 1 13 : 34 : 07 EDT 1983 második sor változtatása
2c
másik sor
.
@@@ you Sat Oct 1 13 : 32 : 28 EDT 1983 egy sor hozzáírása
2d
@@@ you Sat Oct 1 13 : 31 : 03 EDT 1983 új fájl létrehozása
$
```

A szerkesztőparancsok ahhoz, hogy előállítsák a kívánt változatot, végigfutnak a történetfájl elejétől a végéig: az első sorozat a legújabb változatot az eggyel régebbivé, a következő az eggyel régebbit a még eggyel régebbivé alakítja és így tovább. Az új fájlból tehát gyakorlatilag az **ed** futtatásával egyenként előállítjuk a régebbi változatokat.

Biztosan bajba kerülünk, ha a módosítandó fájlban @@@ kezdetű sorok vannak, továbbá a diff(1) **BUGS** szakasza óva int attól, hogy olyan sorokat írjunk, amelyben csak egy pont van. Azért választottuk a szerkesztőparancsok jelölésére a @@@ jelet, mert valószínűtlen, hogy normál szövegben ez előfordul.

Valószínűleg tanulságos volna, ha megmutatnánk, hogyan fejlődtek a **get** és a **put** parancsok, ez azonban elég hosszadalmas lenne, és a különböző változatok bemutatása túl sok elemzést igényelne. Ezért csak a végső változatot mutatjuk be. A **put** az egyszerűbb:

```
# put: történetfájlba helyezés
PATH=/bin:/usr/bin
case $# in
        1)      HIST=$1.H ;;
        *)      echo 'Usage: put file' 1>&2; exit 1 ;;
esac
if test ! -r $1
then
        echo "put: can't open $1" 1>&2
        exit 1
fi
trap 'rm -f /tmp/put.[ab] $$; exit 1' 1 2 15
echo -n 'Summary:'
read Summary
```

```
if get -o /tmp/put.a$$ $1                          # előző változat
then                                               # összeválogatja a részeket
    cp $1 /tmp/put.b$$                             # aktuális változat
    echo "@@@`getname` `date` $Summary" >>/tmp/put.b$$
    diff -e $1 /tmp/put.a$$ >>/tmp/put.b$$         # utolsó különbségek
    sed -n '/^@@@/,$p' <$HIST >>/tmp/put.b$$       # régi különbségek
    overwrite $HIST cat /tmp/put.b$$               # visszaírás
else
    echo "put: creating $HIST"
    cp $1 $HIST
    echo "@@@`getname` `date` $Summary" >>$HIST
fi
rm -f /tmp/put.[ab]$$
```

A **put** az egysoros összefoglalás beolvasása után hívja a **get**-et, hogy a történetfájlból elővegye a fájl előző változatát. A **-o** opció a **get**-nek másik kimeneti fájlt ad meg. Ha a **get** nem találja a történetfájlt, akkor hibastátust ad vissza, és a **put** létrehozza az új történetfájlt. Ha a történetfájl létezik, akkor a **then** ág egy ideiglenes fájlban a legújabb változatból létrehozza az új történetfájlt, amely sorra a legújabb változatból, a @@@ sorból, a legújabb változatot az előzővé alakító szerkesztőparancsokból, valamint a régebbi szerkesztőparancsokból és @@@ sorokból áll. Végül az ideiglenes fájlt az **overwrite** segítségével átmásolja a történetfájlba.

A **get** bonyolultabb, mint a **put**, főleg azért, mert opciói vannak:

```
# get: történetfájlból visszanyerés

PATH=/bin:/usr/bin
VERSION=0
while test "$1" != ""
do
    case "$1" in
        -i) INPUT=$2; shift ;;
        -o) OUTPUT=$2; shift ;;
        -[0-9]) VERSION=$1 ;;
        -*) echo "get: Unknown argument $i" 1>&2; exit 1 ;;
        *) case "$OUTPUT" in
            "") OUTPUT=$1 ;;
            *) INPUT=$1.H ;;
            esac
    esac
    shift
done
OUTPUT=${OUTPUT?"Usage: get [-o outfile] [-i file.H] file"}
INPUT=${INPUT-$OUTPUT.H}
test -r $INPUT || { echo "get: no file $INPUT" 1>&2; exit 1; }
trap 'rm -f /temp/get.[ab]$$; exit 1' 1 2 15
# szétválasztja az aktuális változatot és a szerkesztő parancsokat
sed <$INPUT -n '1,/^@@@/w /tmp/get.a'$$'
                /^@@@/,$w /tmp/get.b'$$
```

```
# végrehajtja a szerkesztéseket
awk </tmp/get.b$$ '
    /^@@@/ { count++ }
    !/^@@@/ && count > 0 && count <= - '$VERSION'
    END { print "$d"; print "w", "'$OUTPUT'" }
' | ed - /tmp/get.a$$
rm -f /tmp/get.[ab]$$
```

Az opciók lényegében a szokásosak. A **-i** és **-o** eltérő bemenetet és kimenetet ad meg, a *-[0-9]* kiválaszt egy adott változatot: a *0* a legutóbbi (alapértelmezés szerinti), *-1* az eggyel előbbi változat és így tovább. Az argumentumokon végigszaladó ciklus egy **while**, amelyben egy **test** és egy **shift** van. Azért nem **for**, mert némelyik opció (**-i**, **-o**) argumentumot használ, ezért szükséges a **shift**. A **for** és a **shift** azonban nem tudnak jól együttműködni, ha a **shift** a **for**-on belül van. Az **ed** mínusz (-) opciója kikapcsolja a fájl írását és olvasását kísérő karakterszámlálást. A

```
test -r $INPUT || { echo "get: no file $INPUT" 1>&2; exit 1; }
```

sor jelentése ugyanaz, mint az

```
if test ! -r $INPUT
then
    echo "get: no file $INPUT" 1>&2
    exit 1
fi
```

(amit a **put** esetében használtunk), de az előző rövidebb és azok számára, akik jól ismerik a || operátort, világosabb is. A { } kapcsos zárójelekben levő parancsokat az aktuális burok, nem pedig egy alburok hajtja végre; ez itt azért kell, hogy az **exit** kilépjen a **get**-ből is, és ne csak az alburokból. A { } karakterek olyanok, mint a **do** és **done** – csak akkor van különleges jelentésük, ha pontosvessző, újsor vagy más parancslezáró után állnak.

Végül elérkeztünk a **get** azon részéhez, amely elvégzi magát a feladatot. Először a **sed** két részre osztja a történetfájlt: a legutolsó változatra és a szerkesztőparancsok halmazára. Ezután az **awk** program feldolgozza a szerkesztőparancsokat. Számlálja (de nem írja ki) a @@@ sorokat, és egészen addig, amíg számuk nem nagyobb, mint a keresett változat száma, lefuttatja a szerkesztőparancsokat (emlékezzünk vissza, hogy az **awk** alapértelmezése az, hogy kiírja a bemenő sorokat). Két **ed** parancsot adunk a történetfájlból kapottakhoz: **$d** parancs törli azt az egyetlen @@@ sort, amelyet a **sed** a legutolsó változatban hagyott; a **w** parancs írja a fájlt végső helyére. Az **overwrite** itt felesleges, mert a **get** csak a fájl változatát változtatja, nem pedig az értékes történetfájlt.

5-29. feladat. Írjunk egy **változat** nevű parancsot, amely két dolgot csinál:

```
$ változat -5 fájl
```

kiírja a történetfájl kiválasztott változatának összefoglalását, a módosítás dátumát és a módosítást végző személyt:

```
$ változat sep 20 fájl
```

megmondja, hogy szeptember 20-án melyik volt a *fájl* aktuális változata! Ezt fel lehet használni pl. így:

```
$ get `változat sep 20 fájl`
```

(a **változat** alkalmas módon visszhangozhatja a történetfájl nevét). □

5-30. feladat. Módosítsuk a **get**-et és a **put**-ot úgy, hogy a történetfájlt külön jegyzékben kezeljék, ne tűzdeljék tele a munkajegyzéket .T fájlokkal! □

5-31. feladat. Általában nem érdemes egy fájl minden változatát megőrizni, ha bizonyos dolgokat már megoldottunk. Hogyan lehet elrendezni, hogy a történetfájl közepéből lehessen változatokat törölni? □

5.10. Visszatekintés

Ha új programot kell írnunk, akkor először arra szoktunk gondolni, hogy hogyan írhatnánk meg azt a kedvenc programnyelvünkön. A mi esetünkben ez a nyelv leggyakrabban a burok.

Szokatlan szintaxisa ellenére a burok egy nagyszerű programnyelv. Mindenképpen magas szintű; operátorai teljes programok. Mivel interaktív, a programokat interaktívan fejleszthetjük, kis lépésekben finomítva, amíg „működnek". Ha a személyes használatnál többre szánjuk programunkat, akkor továbbcsiszolhatjuk, erősíthetjük azt a felhasználók szélesebb rétegei számára. Abban a ritka esetben, amelyben a burokprogram nem elég hatékony, egyes részeit vagy az egészet C nyelvben újraírhatjuk, de tervünk már megvalósult, és működő implementáció van a kezünkben. (A következő fejezetben néhányszor ezt az utat fogjuk követni.)

Ez a fajta megközelítés jellemző a UNIX programozási környezetre – ne kezdjük a semminél! Építsünk arra, amit már mások megcsináltak! Kezdjünk valami kicsivel, és hagyjuk fejlődni! Új ötleteink kipróbálása során használjuk a már meglevő eszközöket!

Ebben a fejezetben sok olyan példát mutattunk, amelyet a meglevő programokkal és a burokkal egyszerűen meg tudtunk csinálni. Gyakran elég volt egyszerűen átrendezni az argumentumokat, gondoljunk például a **cal** esetére! Néha a burok ciklusával megyünk végig fájlnevek egy halmazán vagy parancsok egy sorozatán, mint pl. a **watchfor** vagy a **checkmail** esetében. A bonyolultabb példák is kevesebb munkát adnak, mint a C nyelvben, pl. a mi 20-soros **news** burokváltozatunk egy 350 soros (sic!) C program megfelelője.

Nem az a döntő, hogy van egy programozható parancsnyelvünk, még az sem,

hogy van egy csomó programunk, hanem az, hogy ezek az alkatrészek *együttműködnek*. Azonos módon jelenítik meg és adják tovább az információt. Mindegyiket úgy tervezték, hogy egy feladatra összpontosítson és azt jól csinálja. A burok pedig arra szolgál, hogy ha van egy új ötletünk, akkor annak megvalósítására könnyen és hatékonyan összefűzhessük a már meglevő elemeket. Ez az együttműködés az, amitől a UNIX programozási környezet annyira termelékeny.

Történeti és irodalmi megjegyzések

A **get** és **put** ötlete a Mark Rochkind által alkotott „Source Code Control System"-ből (SCCS) származik (The source code control system, *IEEE Trans. on Software Engineering,* 1975). Az SCCS sokkal hatásosabb és rugalmasabb, mint a mi egyszerű programjaink; nagy programok ipari környezetben való működtetésére készült. Mindazonáltal az SCCS alapja az általunk is használt **diff** program.

6. Programozás a szabványos be- és kimenettel

Eddig a már meglevő eszközök felhasználásával építtetünk újakat, most pedig már elérkeztünk annak határáig, amit a burok, a **sed** és az **awk** alkalmazásával ésszerűen megcsinálhatunk. Ebben a fejezetben néhány egyszerű programot fogunk írni, C programnyelven. Elemzéseinkben és programtervezéseinkben továbbra is az együttműködés gondolata lesz vezérfonalunk – olyan eszközöket akarunk létrehozni, amelyeket mások is használhatnak és tovább tudnak építeni. Minden példánkban ésszerű megvalósítási stratégiát próbálunk bemutatni: elkezdjük egy egyszerű és rövid programmal, ami azonban már valami hasznosat csinál, ezt szükség szerint (csak akkor!) bővítjük újabb tulajdonságokkal és opciókkal.

Jó okunk van arra, hogy az új programokhoz vázlatokat írjunk. Esetenként az adott feladatot nem lehet meglevő programokkal megoldani. A nem szövegfájlokat kezelő programok esetében ez gyakran így van – valójában az eddig bemutatott programok nagy része csak szöveges információt tud jól feldolgozni. A megfelelő méretű és hatékonyságú program létrehozása a burokkal vagy más általános célú eszközzel már túlságosan bonyolult is lehet. A program buroknyelvű változata ilyenkor a programdefiníció finomítására és a felhasználói interfész kialakítására lehet kitűnő eszköz. (És ha jól működik, miért csináljuk újra?) Jó példa erre az előző fejezet **zap** programja: csak néhány percbe telt a burokban az első változatot megírni; a végleges változat megfelelő felhasználói interfésszel rendelkezik, de túlságosan lassú.

C nyelven fogunk programozni, mert ez a UNIX rendszerek szabványos nyelve – a vezérlőprogram és az összes felhasználói program C nyelven készült –, a UNIX más nyelvet közel sem támogat ennyire. Feltételezzük, hogy Olvasóink ismerik a C nyelvet, legalábbis annyira, hogy követni tudják. Ha nem, ajánljuk B. W. Kernighan és D. M. Ritchie: A C programozási nyelv című könyvét.*

Használni fogjuk még a „szabványos be- és kimenet könyvtárát", amely a C nyelvű programok számára hatékony és rugalmas be- és kimenetet és rendszerszolgáltatásokat nyújtó rutinok gyűjteménye. A szabványos be- és kimenet könyvtára sok, a C nyelvet támogató nem UNIX rendszeren is hozzáférhető. Így azok a programok, amelyek a C nyelv lehetőségein belül bonyolítják rendszerkapcsolataikat, könnyen átvihetők ezekre a rendszerekre is.

* A könyv magyarul is megjelent (Műszaki Könyvkiadó, 1985).

Fejezetünk példáinak van egy közös tulajdonsága: mindegyik általunk rendszeresen használt apró eszköz, amely a 7. kiadásnak nem része. Ha Olvasóink rendszerében vannak hasonló programok, a megoldások összehasonlítása hasznos lehet. Ha pedig nincsenek, új eszközként ugyanolyan hasznosak lehetnek, mint nekünk voltak. Abban mindenképpen segítenek példaprogramjaink, hogy belássuk: nincs tökéletes rendszer, azonban gyakran egyszerű javítani a programokat és eltüntetni a hibákat.

6.1. Szabványos bemenet és kimenet: vis

Sok program csak egy bemenetről olvas és egy kimenetre ír. Ilyen programokban tökéletesen megfelelő lehet a csak szabványos bemenet és kimenet használata, és ez szinte mindig elég az induláshoz.

Bemutatunk egy **vis** nevű programot, amely szabványos bemenetét szabványos kimenetére másolja, de az összes nem nyomtatható karaktert láthatóvá teszi azáltal, hogy \nnn formában kiírja, ahol nnn a karakter oktális értéke. A **vis** felbecsülhetetlen segítséget nyújt a fájlokba valahogyan belekerülő idegen és nemkívánatos karakterek felkutatásában. Pl. a **vis** minden visszalépés- (backspace) karaktert \010 formában ír ki, ez a visszalépés-karakter oktális értéke:

```
$ cat x
abc
$ vis <x
abc\010\010\010__
$
```

Több fájlt a **vis** kezdetleges változatával úgy nézhetünk végig, hogy a **cat** felhasználásával összegyűjtjük a fájlokat:

```
$ cat file1 file2 ... / vis
...
$ cat file1 file2 ... / vis / grep '\\'
...
```

és így nem kell megtanulni azt, hogy hogyan lehet egy programból fájlokhoz hozzáférni.

Egyébként úgy tűnhet, hogy ezt a feladatot a **sed**-del is meg lehet oldani, hiszen az **l** parancs érthető formában kiírja a nemnyomtatható karaktereket:

```
$ sed -n l x
abc←←←__
$
```

A **sed** kimenet valószínűleg világosabb, mint a **vis** kimenete, de a **sed**-et nem szöveges fájlokra hiába próbáljuk alkalmazni:

```
$ sed -n l /usr/you/bin
$
```
Semmit sem ír ki!

(Ez volt a PDP-11-en; az egyik VAX rendszeren a **sed** abortált, valószínűleg azért, mert bemenete úgy nézett ki, mintha egyetlen nagyon hosszú szövegsor lenne.) Tehát a **sed** nem megfelelő, arra kényszerülünk, hogy új programot írjunk. A legegyszerűbb bemeneti és kimeneti rutinok a **getchar** és a **putchar**. A **getchar** minden hívásra behozza a szabványos bemenetről a következő karaktert. A szabványos bemenet lehet egy fájl, egy cső vagy (alapértelmezés szerint) a terminál – a program nem is tudja, hogy most éppen melyiket használja. Hasonlóan a **putchar(c)** a c karaktert a szabványos kimenetre írja, alapértelmezésben ez is a terminál.

A **printf(3)** függvény a kimeneten formátumátalakítást végez. A **printf** és **putchar** hívások bármilyen sorrendben lehetnek, a kimenet a hívások sorrendjében jelenik meg. A bemenet formátumának átalakítására a megfelelő függvény: a **scanf(3).** Ez a szabványos bemenetet olvassa és szükség szerint karakterláncokká, számokká stb. tördeli azt. A **scanf** és a **getchar** hívások is keverhetők.

A **vis** első változata:

```
/* vis: a furcsa karaktereket láthatóvá teszi (1. változat) */
# include <stdio.h>
# include <ctype.h>

main ( )
{
  int c;

  while ((c = getchar()) != EOF)
    if (isascii(c) &&
      (isprint(c) || c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else
      printf("\\%030", c);
  exit(0);
}
```

A **getchar** a bemenet következő bájtját adja, kivéve, ha fájl vége (vagy hiba) van, akkor EOF értéket ad. Egyébként az EOF nem a fájl egy bájtja, emlékezzünk vissza a fájl végéről a 2. fejezetben mondottakra! Az EOF értéke garantáltan különbözik minden olyan értéktől, amely egy bájtban előfordulhat, ezért mindig megkülönböztethető az igazi adattól; a **c** deklaráció szerint **int**, nem **char** típusú, így elég nagy ahhoz, hogy az EOF értéke beleférjen. A

```
#include <stdio.h>
```

sornak minden forrásfájl elején meg kell jelennie. A C fordítóprogramja ennek hatására olvassa a szabványos rutinokhoz és szimbólumokhoz a fejléc fájlját (**/usr/include/stdio.h**), amely tartalmazza az EOF definícióját. A **<stdio.h>** nevet a teljes fájlnév rövidítésére használjuk.

A **<ctype.h>** karaktervizsgáló makrói

isalpha(c)	alfabetikus: a–z A–Z
isupper(c)	nagybetű: A–Z
islower(c)	kisbetű: a–z
isdigit(c)	számjegy: 0–9
isxdigit(c)	hexadecimális számjegy: 0–9 a–f A–F
isalnum(c)	alfabetikus vagy számjegy
isspace(c)	szóköz, tabulátor, újsor, függőleges tabulátor, lapdobás, return
ispunct(c)	nem alfanumerikus, nem vezérlő, nem szóköz
isprint(c)	nyomtatható: bármilyen grafikus
iscntrl(c)	vezérlőkarakter: 0 <= c < 040 || c == 0177
isascii(c)	ASCII karakter: 0 <= c <= 0177

A **<ctype.h>** a **/usr/include** egy másik fejléc fájlja, amely a karakterek tulajdonságainak megállapítására szolgáló gépfüggetlen teszteket tartalmaz.

Mi az **isascii** és **isprint** rutinokkal határozzuk meg, hogy a bemenő karakter ASCII (vagyis értéke kevesebb, mint 0200) és nyomtatható karakter-e; az egyéb teszteket a 6.1. táblázatban soroltuk fel. Megjegyezzük, hogy az újsor, a tabulátor és a szóköz a **<ctype.h>** definíciója szerint nem „nyomtatható" karakterek.

A **vis** végén az **exit** hívás a program helyes működését nem befolyásolja, de a program végén ez biztosítja a program meghívója számára a normál kilépési státust (megállapodás szerint 0-t). Egy másik lehetőség a státusz visszaadására, hogy a **main**-t **return 0** utasítással hagyjuk el; így a **main** által visszaadott érték lesz a program kilépési státusa. Ha nincs explicit **return** vagy **exit**, akkor kilépési státus értéke kiszámíthatatlan.

Egy C nyelvű programot úgy fordítunk, hogy a forrást betesszük egy .c-re végződő nevű fájlba (mint pl. **vis.c**), lefordítjuk a **cc**-vel, majd a fordító által az **a.out** nevű fájlban hagyott eredményt futtatjuk (az *a* betű az assemblert jelzi):

```
$ cc vis.c
$ a.out
jó reggeltctl-g      A zárókarakter: ctl-g.
jó reggelt\007
ctl-d
$
```

Általában, ha már működik a programunk, akkor nevezzük át az **a.out**-ot, vagy használjuk a **cc** fordítóprogram **-o** opcióját, amely közvetlenül ezt teszi:

```
$ cc -o vis vis.c      Kimenet a vis-ben, nem az a.out-ban.
```

6-1. feladat. A **vis** programunk tervezésekor úgy döntöttünk, hogy a tabulátorokat békén hagyjuk, nem tesszük azokat \011, vagy →, vagy \t alakban láthatóvá, ugyanis a **vis** programot főleg arra használjuk, hogy a valóban rendellenes karaktereket megmutassa. Egy másik tervezési elképzelés a kimenet *minden* karakterét – tabulátort, nemgrafikus karaktert, sorvégi szóközt stb. – egyértelműen azonosítja. Módosítsuk a **vis**-t ennek megfelelően úgy, hogy a tabulátor a fordított törtvonal, a visszalépés, a lapdobás stb. karaktereket megszokott C reprezentációjukkal, \t, \\, \b, \f stb. írja ki! Meg tudjuk ezt csinálni egyértelműen? Hasonlítsuk össze tervünket a

```
$ sed -n l
```

paranccsal! □

6-2. feladat. Módosítsuk a **vis**-t úgy, hogy a túl hosszú sorokat elfogadható hosszúságúakra tördelje! Hogyan hat ez az előző feladatban kívánt egyértelmű kimenetre? □

6.2. Programargumentumok: a vis második változata

A C program végrehajtásakor a parancssor-argumentumok a **main** függvény számára egy **argc** szám és egy **argv** mutatótömb formájában érhetők el. Az utóbbi az argumentumokat tartalmazó karakterláncokra mutat. Megállapodás szerint az **argv[0]** maga a parancs neve, így a karakterláncok száma, az **argc** mindig nagyobb, mint 0; a „hasznos" argumentumok: az **argv[1]...argv[argc − 1].** Emlékezzünk vissza, hogy a < és > átirányítást a burok végzi, nem az egyes programok, így az átirányítás nem befolyásolja azt, hogy a program hány argumentumot lát!

Az argumentumkezelés bemutatására bővítsük a **vis** programot egy opcionális argumentummal: a **vis -s** dobja el a nemnyomtatható karaktereket ahelyett, hogy láthatóvá tenné őket. Ez az opció nagyón hasznos, amikor más rendszerekből kapott fájlokat meg akarunk tisztítani a nálunk felesleges karakterektől. Pl. olyan fájlok esetén, amelyek a sorok lezárására újsor helyett a *CRLF*-et (kocsivissza és soremelés) használják.

```
/* vis: a furcsa karaktereket láthatóvá teszi (2. változat) */
#include <stdio.h>
#include <ctype.h>
main(argc, argv)
  int argc;
  char *argv[];
{
  int c, strip = 0;
  if (argc > 1 && strcmp(argv[1], "-s") == 0)
    strip = 1;
  while ((c = getchar()) != EOF)
```

```
    if (isascii (c) &&
      (isprint(c) || c=='\n'|| c=='\t' || c=' '))
      putchar(c);
    else if (!strip)
      printf("\\%03o", c);
  exit (0);
}
```

Az **argv** mutató egy olyan tömbre mutat, amelynek egyes elemei szintén tömbre, karakterek tömbjére mutatnak; minden tömböt egy ASCII **NUL** karakter (**'\0'**) zár le, ezért az karakterláncként kezelhető. A **vis** fenti változata először megnézi, hogy van-e argumentum és ha van, akkor az **−s**-e. (Az érvénytelen argumentumokat

6.2. táblázat

Karakterláncokra vonatkozó szabványos függvények

strcat(s,t)	az s karakterlánchoz hozzáfűzi a t karakterláncot; s-et ad vissza
strncat(s,t,n)	az s karakterlánchoz hozzáfűzi a t karakterlánc legfeljebb n karakterét
strcpy(s,t)	a t-t az s-re másolja; s-et ad vissza
strncpy(s,t,n)	pontosan n karaktert másol, ha szükséges, nullával tölti fel
strcmp(s,t)	összehasonlítja s-et és t-t; a $<$, $==$, $>$ relációkra <0, 0, >0 értéket ad vissza
strncmp(s,t,n)	legfeljebb n karaktert hasonlít össze
strlen(s)	s hosszát adja vissza
strchr(s,c)	az s-ben levő első c-re mutatót ad, ha nincs, NULL-t ad vissza
strrchr(s,c)	az s-ben levő utolsó c-re mutatót ad, ha nincs, NULL-t ad vissza. Régebbi rendszerekben az **index** és az **rindex** csinálják ezt
atoi(s)	s egészértékét adja
atof(s)	s lebegőpontos értékét adja; kell hozzá a **double atof()** deklaráció
malloc(n)	n bájtnyi tárrészre vonatkozó mutatót ad vissza, NULL-t, ha nincs
calloc(n,m)	n × m kinullázott bájtra vonatkozó mutatót ad vissza, ha nincs NULL-t a **malloc** és **calloc** char *-ot ad vissza
free(p)	a **malloc** és **calloc** által lefoglalt tárterületet felszabadítja

elhagyja.) A **strcmp (3)** függvény összehasonlít két karakterláncot, és nullát ad vissza, ha azok azonosak.

A 6.2. táblázat néhány karakterlánc-kezelő és általános célú segédfüggvényt foglal össze, közülük az egyik az **strcmp.** Általában célszerűbb ezeket a függvényeket használni, mint újat írni, mivel ezek szabványosak, ellenőrzöttek, és gyakran gyorsabbak, mint amit mi magunk tudnánk írni, ugyanis egyes gépekre optimalizáltak (gyakran assembly nyelven írottak).

6-3. feladat. Változtassuk meg a **−s** argumentumot úgy, hogy a **vis −s***n* csak az *n* számú vagy annál több egymás utáni nyomtatható karakterből álló sorozatokat írja ki, a nemnyomtatható karaktereket és a rövidebb nyomtatható sorozatokat hagyja el! Ez nagyon hasznos arra, hogy a nem szöveges fájlokban (pl. végrehajtható programokban) megkeressük a szöveges részeket. A rendszer néhány változatában van egy **strings** program, amely éppen ezt teszi. Melyik a jobb, ha van egy külön program vagy ha a **vis**-nek van egy argumentuma? □

6-4. feladat. A C nyelvű forráskód használata a UNIX rendszer egyik erőssége − a C nyelv elegáns megoldásokat kínál sok programozási feladathoz. Elemezzük a C forráskód olvashatóságából, ill. az assembly nyelvű optimalizálásból származó előnyöket és hátrányokat! □

6.3. Fájlelérés: a vis harmadik változata

A **vis** első két változata a szabványos bemenetről olvas és a szabványos kimenetre ír, mindkettőt a buroktól örökölte. A következő módosítás, hogy a **vis** név szerint érje el a fájlokat: tehát a

```
$ vis fájl1 fájl2 ...
```

a szabványos bemenet helyett a megnevezett fájlokon fusson végig. Ha nincs fájlnév-argumentum, akkor továbbra is azt akarjuk, hogy a **vis** a szabványos bemenetet olvassa.

Az első kérdés, hogy hogyan férhetünk hozzá az olvasandó fájlokhoz − azaz hogyan kapcsolhatjuk a fájlneveket az adatot valójában beolvasó utasításokhoz.

A szabályok egyszerűek. Mielőtt egy fájlt olvashatnánk vagy írhatnánk, meg kell nyitnunk azt az **fopen** szabványos könyvtári függvénnyel. Az **fopen** veszi a fájlnevet (pl. **múlandó** vagy **/etc/passwd**), elvégzi a kötelező adminisztrációt és egyezőségvizsgálatot, majd visszaadja a további fájlműveletekben használandó belső nevet.

A belső név lényegében egy mutató, a neve *fájlmutató,* amely a fájllal kapcsolatos információkat tartalmazó adatszerkezetre mutat. Ebben van feljegyezve a puffertár helye, az aktuális karakter helye a puffertárban, az olvasás- vagy írásjelző és más hasonlók. Az **<stdio.h>** meghívásával előállított definíciók egyike egy **FILE** nevű szerkezet definíciója. A fájlmutatót a következőképpen deklarálhatjuk:

```
FILE *fp;
```

ami azt jelenti, hogy az **fp** egy **FILE**-ra vonatkozó fájlmutató. Az **fopen** egy **FILE**-ra vonatkozó fájlmutatót ad vissza; az **fopen**-re egy típusdeklarációja az **<stdio.h>**-ban található.

Az **fopen**-hívás a programban a következő:

```
char *name, *mode;
fp=fopen(name, mode);
```

Az **fopen** első argumentuma a fájlnév, amely egy karakterlánc. A második argumentum szintén egy karakterlánc, amely jelzi a fájl használatának módját. A megengedett módok: olvasás (**"r"**); írás (**"w"**); és hozzátétel (**"a"**).

Ha egy fájl, amit írásra vagy hozzátételre nyitunk meg, nem létezik, akkor a rendszer létrehozza azt, hacsak lehetséges. Ha egy meglevő fájlt újra írásra nyitunk meg, akkor a régi tartalma elvész. Hibát okoz, ha nem létező fájlt próbálunk olvasni, vagy ha engedély nélkül akarunk egy fájlt írni vagy olvasni. Ha hiba van, akkor **fopen** érvénytelen NULL mutató értéket ad vissza (ezt a **<stdio.h>**-ban általában **(char *)0**-ként definiáljuk.)

Ezek után tudnunk kell a már megnyitott fájlt olvasni vagy írni! Számos lehetőségünk közül a legegyszerűbbek a **getc** és a **putc**. A **getc** veszi a fájlból a következő karaktert.

```
c = getc(fp)
```

az **fp** által meghatározott fájlból teszi a következő karaktert **c**-be; **EOF**-ot ad vissza, ha a fájl végére ér. A **putc** analóg ezzel:

```
putc(c, fp)
```

az **fp** fájlba következő karakterként beteszi a **c**-t és **c**-t ad vissza. Hiba esetén a **getc** és a **putc** is **EOF**-ot ad.

Ha egy programot indítunk, akkor három fájl máris nyitva van, és megvannak a fájlmutatóik is. Ezek a fájlok a szabványos bemenet, a szabványos kimenet és a szabványos hibakimenet; a nekik megfelelő fájlmutatók sorrendben az **stdin, stdout** és **stderr**. Ezek a mutatók az **<stdio.h>**-ban vannak deklarálva és mindenütt használhatók, ahol **FILE *** használható. Ezek állandók és nem változók, így nem lehet értéket adni nekik.

A **getchar()** ugyanaz, mint a **getc(stdin)**, a **putchar(c)** és a **putc(c,stdout)**. Valójában e négy „függvény" mindegyike makróként van definiálva az **<stdio.h>**-ban. Azért makrók, mert így gyorsabban futnak, mintha minden karakterre volna egy függvényhívás. A 6.3. táblázatban az **<stdio.h>** néhány további definícióját soroljuk fel.

Az akadályokon túljutva most már megírhatjuk a **vis** harmadik változatát, amelynek argumentumkezelése a következő: ha vannak parancssor-argumentumok, akkor azokat sorban feldolgozza; ha nincsenek argumentumok, akkor a szabványos bemenetet dolgozza fel.

Néhány **<stdio.h>** definíció

stdin	szabványos bemenet
stdout	szabványos kimenet
stderr	szabványos hiba
EOF	fájl vége; általában −1
NULL	érvénytelen mutató; általában 0
FILE	fájlmutatók deklarálására használjuk
BUFSIZ	normál be- és kimeneti bufferméret (rendszerint 512 vagy 1024)
getc(fp)	az fp áradatból egy karaktert visszaad
getchar()	**getc(stdin)**
putch(c,fp)	a c karaktert az fp áradatba teszi
putchar(c)	**putc(c, stdout)**
feof(fp)	nemnulla, ha az fp áradaton fájl vége van
ferror(fp)	nemnulla, ha az fp áradatban hiba van
fileno(fp)	az fp áradatfájl leírása, l. a 7. fejezetet

```
/* vis: a furcsa karaktereket láthatóvá teszi: (3. változat) */

#include <stdio.h>
#include <ctype.h>
int strip = 0;          /* 1 =>különleges karaktereket eldobja */
main(argc, argv)
  int argc;
  char *argv[];
{
  int i;
FILE *fp;

while (argc >1 && argv[1][0] == '-') {
  switch (argv[1][1]) {
  case 's':     /* -s: furcsa karaktereket átugorja */
    strip = 1;
    break;
  default:
    fprintf(stderr, "%s: unknown arg %s\n",
      argv[0], argv[1]);
    exit(1);
  }
  argc--;
  argv++;
}
```

```
  if (argc == 1)
    vis (stdin);
  else
    for (i = 1; i < argc; i++)
    if ((fp=fopen(argv[i], "r")) == NULL) {
        fprintf(stderr, "%s: can't open %s\n",
          argv[0], argv[i]);
        exit(1);
    } else {
        vis(fp);
        fclose(fp);
    }
  exit(0);
}
```

Ez a program kihasználja, hogy az opcionális argumentumok jönnek először. Az egyes opcionális argumentumok feldolgozása után az **argc** és **argv** értékeket átállítja úgy, hogy a program többi része független legyen a feldolgozott argumentumnak a lététől. Bár a **vis** csak egyetlen opciót ismer fel, a programot ciklusba szervezve írtuk, hogy bemutassuk az argumentumfeldolgozás szervezésének ezt a lehetőségét is. Az első fejezetben felhívtuk a figyelmet arra, hogy a UNIX programok rendetlenül kezelik az opcionális argumentumokat. Ennek egyik oka – az anarchián kívül – az, hogy nyilvánvalóan könnyű olyan kódot írni, amely mindenféle argumentumelemzést végez. A néhány rendszerben megtalálható **getopt(3)** függvény kísérlet a helyzet ésszerűsítésére, érdemes megnézni, mielőtt a saját változatot írnánk.

A **vis** rutin egyetlen fájlt ír ki:

```
vis(fp) /* láthatóvá teszi a FILE *fp karaktereit */
  FILE *fp;
{
  int c;

  while ((c = getc(fp)) != EOF)
    if (isascii(c) &&
      (isprint(c) || c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else if (!strip)
      printf("\\%03o", c);
}
```

Az **fprintf** függvény ugyanaz, mint a **printf,** de argumentumában fájlmutató jelöli ki, hogy melyik fájlra írjon.

Az **fclose** függvény megszünteti az **fopen** függvénnyel a fájlmutató és a külső név között létrehozott kapcsolatot, felszabadítja a fájlmutatót egy másik fájl számára. Miután egy programban az egyszerre megnyitható fájlok száma korlátozott (kb. 20), legjobb azonnal felszabadítani a fájlokat, ha már nincs rájuk szükség. Normál esetben a **printf, putc** stb. szabványos könyvtári függvények kimenete pufferelt, tehát nagy adagokban hatékonyan írhatnak. (Kivétel a terminálkimenet, amelyre rendszerint

karakterenként, de legalábbis soronként írnak.) A kimeneti fájlra alkalmazott **fclose** kiviszi a pufferben maradt kimenetet. Ha egy program az **exit**-re hívja vagy a **main**-ből tér vissza, akkor automatikusan minden megnyitott fájlra hívja az **fclose**-t is.

Az **stderr** ugyanúgy hozzá van rendelve a programokhoz, mint az **stdin** és az **stdout.** A **stderr**-re írt kimenet még akkor is a felhasználói terminálon jelenik meg, ha a kimenetet átirányítottuk. A **vis** a diagnosztikát az **stderr**-re írja, nem az **stdout**-ra, így ha valamilyen okból egy fájl nem elérhető, az üzenet akkor is megtalálja a felhasználói terminálra vezető utat, és nem tűnik el egy csőhálózatban vagy kimeneti fájlban. (A szabványos hibát valamivel a csövek után találták ki, éppen azért, mert a hibaüzenetek kezdtek eltűnni a csőhálózatban.)

Kicsit önkényesen úgy döntöttünk, hogy a **vis** program megszakítja a futását, ha a bemeneti fájlt nem tudja megnyitni; ez indokolt egy olyan programnál, amelyet általában interaktívan és egy bemeneti fájllal használunk. Esetenként természetesen indokolt lehet más megoldás is.

6-5. feladat. írjunk egy **nyomtatható** nevű programot, amely kiírja azoknak az argumentumfájloknak a nevét, amelyek csak nyomtatható karakterekből állnak; ha egy fájlban van nemnyomtatható karakter is, akkor annak nevét nem írja ki!

A **nyomtatható** jól használható pl. a következő helyzetben:

```
$ pr `nyomtatható *` | lpr
```

Bővítsük programunkat egy **-v** opcióval, amely megfordítja a teszt értelmét, ugyanúgy, mint a **grep**-nél! Mit csináljon a **nyomtatható,** ha nincs fájlnévargumentuma? Milyen státust kell a **nyomtatható**-nak visszaadnia? □

6.4. Képernyőnyomtató: p

Eddig a **cat** segítségével vizsgáltunk fájlokat. Ha a fájl túl hosszú és a rendszerhez nagy sebességű vonallal csatlakozunk, akkor a **cat** olyan gyorsan adja a kimenetet, hogy nem tudjuk elolvasni, még akkor sem, ha elég gyorsak vagyunk a *ctl−s* és a *ctl−q* nyomkodásával.

Nyilvánvalóan kell egy olyan program is, amely kis, ellenőrizhető részekben írja ki a fájlt. Ilyen szabványos program azonban nincs, valószínűleg azért, mert az eredeti UNIX rendszert még a papírra író (hard-copy) terminálok és lassú kommunikációs vonalak idejében írták. Ezért következő példánk, a **p** nevű program képernyőnyi egységenként írja ki a fájlt. Minden egység kiírása után a felhasználó engedélyére vár, és csak az engedélyezés után folytatja a következővel. (**p** egy szép, rövid név egy olyan programnak, amelyet gyakran használunk.) Ugyanúgy, mint a többi program, a **p** is az argumentumként megadott fájlokat vagy a szabványos bemenetet olvassa:

```
$ p vis.c
...
```

```
$ grep '#define' *.[sz] | p
...
$
```

Ezt a programot a legjobb C nyelven megírni, ugyanis így egyszerű, a burokkal pedig nehéz; a szabványos eszközök nem jók akkor, ha a fájlból vagy csőhálózatból jövő bemenetet a terminálbemenettel keverjük.

A trükkmentes alapmegoldás az, hogy a bemenetet kis egységekben írjuk ki. Célszerű egységhossz a 22 sor: ez alig kevesebb, mint a legtöbb terminál képernyőjén lévő sorok száma (24), és a szabványos 66 soros nyomtatványoldal harmada. A program a legegyszerűbben úgy várhat a felhasználó válaszára, hogy a 22 soros egység utolsó újsor karakterét nem írja ki. A kurzor így a sor jobb szélén áll meg ahelyett, hogy visszaugrana a bal margóra. Ha a felhasználó megnyomja a *RETURN*-t, ezzel megadja a hiányzó újsort, és így a következő sor a megfelelő helyen megjelenik. Ha a felhasználó *ctl—d* vagy *q*-t ír a képernyő végén, akkor a **p** program befejezi működését.

A hosszú sorokat nem figyeljük és nem törődünk a többszörös fájlokkal sem: minden megjegyzés nélkül átugrunk az egyikről a másikra. Ily módon a

```
$ p fájlnevek...
```

parancssor hatása ugyanaz lesz, mint a

```
$ cat fájlnevek... | p
```

parancssoré. Ha fájlneveket is ki akarjuk íratni, azokat egy ilyen **for** ciklussal hozzátehetjük:

```
$ for i in fájlnevek...
> do
>     echo $i:
>     cat $i
> done | p
```

Nagyon sok tulajdonsággal bővíthetjük ezt a programot. Jobb, ha egy egyszerűsített változattal kezdünk, és azután hagyjuk a kísérletek eredményeként fejlődni. Ily módon valóban azokkal a tulajdonságokkal fog rendelkezni, amelyekre az embereknek szükségük van, nem pedig azokkal, amelyekről mi úgy gondoljuk, hogy kellhetnek nekik.

A **p** alapszerkezete ugyanaz, mint a **vis**-é; a főrutin végigfut a fájlokon, mindegyikre meghívja a munkát végző **print** rutint.

```
/* p: egységenként kiírja a bemenetet (1. változat) */

#include <stdio.h>
#define PAGESIZE 22
char *progname; /* programnév a hibaüzenetekhez */
```

```
main(argc, argv)
  int argc;
  char *argv[];
{
  int i;
  FILE *fp, *efopen();
  progname = argv[0];
  if (argc == 1)
    print(stdin, PAGESIZE);
  else
    for (i = 1; i < argc; i++) {
      fp = efopen(argv[i], "r");
      print(fp, PAGESIZE);
      fclose(fp);
    }
  exit(0);
}
```

Az **efopen** rutin egy nagyon hétköznapi műveletet csinál: megpróbálja megnyitni a fájlt; ha nem lehet, hibaüzenetet ír és kilép. A hibaüzenetet adó progam azonosítására az **efopen** a program nevét tartalmazó **progname** külső karakterláncra hivatkozik, amely értékét a **main**-ben kapja meg.

```
FILE *efopen(file, mode)  /* fopen a fájlra, ha nem lehet, kilép */
  char *file, *mode;
{
  FILE *fp, *fopen();
  extern char *progname;

  if ((fp = fopen(file, mode)) != NULL)
    return fp;
  fprintf(stderr, "%s: can't open file %s mode %s\n",
    progname, file, mode);
  exit(1);
}
```

Néhány más **efopen**-változatot is kipróbáltunk, mielőtt emellett döntöttünk. Az egyik az üzenet kiírása után visszatért, nullmutatóval jelezve a hibát. Ez a hívónak választási lehetőséget adott, hogy folytassa a programot vagy kilépjen belőle. Egy másik tervezet szerint az **efopen**-nek egy harmadik argumentumban meg lehet adni, hogy visszatérjen-e, ha nem sikerült egy fájlt megnyitnia. Majdnem minden példánkban értelmetlen a program folytatása, ha egy fájlt nem lehet megnyitni, tehát a mi céljainkra az **efopen** jelenlegi változata a legmegfelelőbb.

A **p** program munkájának igazi részét a **print** végzi:

```
print(fp, pagesize)  /* fp-t pagesize egységekben írja ki */
  FILE *fp;
  int pagesize;
{
```

```
  static int lines = 0; /* az eddigi sorok száma */
  char buf[BUFSIZ];

  while (fgets(buf, sizeof buf, fp) != NULL)
    if (++lines < pagesize)
      fputs(buf, stdout);
    else {
      buf[strlen(buf)-1] = '\0';
      fputs(buf, stdout);
      fflush(stdout);
      ttyin();
      lines = 0;
    }
}
```

A bemeneti puffer méreteként a **<stdio.h>**-ban megadott **BUFSIZ**-ot használtuk. Az **fgets(buf, size, fp)** az **fp**-ből a következő újsor-karakterig tartó sort, az újsor-karaktert is beleértve a **buf**-ba teszi és hozzárak egy lezáró **\0**-t, de legfeljebb **size − 1** számú karaktert másol. Fájl végén **NULL**-t ad vissza. (Az **fgets** lehetne jobban is tervezve: a **buf**-ot adja vissza és nem a karakterszámot; továbbá nem figyelmeztet, ha a bemenő sor túl hosszú. Nem veszíti el a karaktereket, de meg kell néznünk a **buf**-ban, hogy valójában mi is történt.)

A **strlen** függvény a karakterlánc hosszát adja meg. Arra használjuk, hogy az utolsó bemenő sor végéről elhagyjuk az újsort. Az **fputs(buf, fp)** a **buf** karakterláncot átírja az **fp** fájlba. A lap végén az **fflush** hívás kikergeti az összes pufferezett kimenetet.

Az egyes lapok kiírása után a felhasználótól érkező válasz olvasása a **ttyin** rutin feladata. A **ttyin** nem olvashatja a szabványos bemenetet, hiszen a **p**-nek akkor is működnie kell, ha bemenete egy fájlból vagy csőhálózatból jön. Ennek kezelésére a program megnyitja a **/dev/tty** fájlt, amely a szabványos bemenet átirányításától függetlenül a felhasználói terminál. A **ttyin**-t úgy írtuk meg, hogy a válasz első karakterét adja vissza, bár ezt a tulajdonságát itt nem használjuk ki.

```
ttyin() /* a /dev/tty-ról érkező válasz feldolgozása 1. változat) */
{
  char buf[BUFSIZ];
  FILE *efopen();
  static FILE *tty = NULL;

  if (tty == NULL)
    tty = efopen("/dev/tty", "r");
  if (fgets(buf, BUFSIZ, tty) == NULL || buf[0] == 'q')
    exit(0);
  else              /* közönséges sor */
    return buf[0];
}
```

A **dev/tty** fájlmutatót **static**-nak deklaráltuk, ezért a **ttyin** hívások között megőrzi az értékét; a **/dev/tty** fájl csak az első hívástól lesz nyitva.

Nyilvánvalóan vannak további tulajdonságok, amelyeket kevés munkával hozzáírhatnánk a **p**-hez, de érdemes megjegyezni, hogy a programunk első változata pontosan azt csinálta, amit leírtunk: kiírt 22 sort és várt. Sok idő eltelt, mire valamit hozzáírtunk, és mind ez ideig nagyon kevesen használták ezeket az extra tulajdonságokat.

A **p** program egy egyszerű bővítéseként legyen az oldalankénti sorok száma egy **pagesize** nevű változóban, amelynek értékét a parancssorból beállíthatjuk:

```
$ p -n ...
```

n-soros egységeket ír ki. Ehhez csak néhány ismerős sort kell a **main** elejéhez írni:

```
/* p: egységenként kiírja a bemenetet (2. változat) */
  ...
  int i, pagesize = PAGESIZE;

  progname = argv[0];
  if (argc > 1 && argv[1][0] == '-') {
    pagesize = atoi(&argv[1][1]);
    argc--;
    argv++;
  }
  ...
```

Az **atoi** függvény a karakterláncot egész számmá alakítja (l. **atoi(3)**).

A **p** egy másik egyszerű kiegészítése lehet, hogy az oldal végén ideiglenesen kiléphessen valamilyen más parancsot végrehajtani. Az **ed**-del és sok egyéb programmal analóg módon, ha a felhasználó felkiáltójellel kezdődő sort ír, akkor a sor további részét a program parancsnak értelmezi és végrehajtásra átadja a buroknak. Ennek megvalósítása is triviális, ugyanis van egy **system(3)** nevű függvény, amely pontosan ezt csinálja. A **ttyin** módosított változata a következő:

```
ttyin() /* a /dev/tty-ról érkező válasz feldolgozása (2. változat) */
{
  char buf[BUFSIZ];
  FILE *efopen();
  static FILE *tty = NULL;

  if (tty == NULL)
    tty = efopen("/dev/tty", "r");
  for (;;) {
    if (fgets(buf, BUFSIZ, tty) == !! buf[0] == 'q')
    exit (0); {
    else if (buf[0] == '!') {
      system(buf+1);
      printf("!\n");                    /* itt a HIBA */
    }
    else                                /* normál sor */
      return buf[0];
  }
}
```

Sajnos a **ttyin** ezen változatának van egy titokzatos, rosszindulatú hibája. A **system** által futtatott parancs örökli **p**-től a szabványos bemenetet, így ha **p** csőhálózatból vagy fájlból olvas, akkor a parancs kölcsönhatásba kerül annak bemenetével:

```
$ cat /etc/passwd | p -1
root:3D.fHR5KoB.3s:0:1:s.User:/:/ed
?
!
```

p-n belül az **ed**-et hívja
ed az **(etc/passwd**-t olvassa, ...
... összezavarodik és kilép

A megoldáshoz ismernünk kellene, hogy a UNIX hogyan vezérli a folyamatokat, de ezt csak a 7.4. szakaszban ismertetjük. Most elégedjünk meg azzal, hogy a szabványos könyvtári **system** bajt okozhat, de a **ttyin** jól működik, ha a 7. fejezet **system**-változatával fordítjuk le.

Eddigi két programunk, a **vis** és a **p** a **cat** változatainak tekinthetők, néhány díszítéssel. Ne legyenek ezek a **cat** új részei, amelyeket a **-v** vagy **-p** opcionális argumentumokkal érhetünk el? A kérdés az, hogy új programot írjunk-e vagy a régit egészítsük ki újabb és újabb tulajdonságokkal, amint új igények merülnek fel. Nincs határozott válaszunk, de van néhány elvünk, amelyek segíthetnek a döntésben.

Alapelvünk, hogy egy programnak csak egy fő feladata legyen – minél többet csinál, annál nagyobb, lassabb lesz, nehezebb karbantartani és alkalmazni. Ráadásul a tulajdonságok gyakran kihasználatlanul hevernek, mert az emberek nem emlékeznek az opciókra.

Ez azt sugallja, hogy ne kapcsoljuk össze a **vis** és a **cat** programokat. A **cat** csak változatlanul lemásolja a bemenetét, míg a **vis** átalakítja. Összekapcsolódásukkal egy olyan programot kapnánk, ami két különböző dolgot csinál. Majdnem ugyanez a helyzet a **cat** és **p** összekapcsolásával is. A **cat** gyors, hatékony másolásra készült, a **p** nézelődésre. A **p** átalakítja a kimenetét: minden 22-ik újsor-karakterét eldobja. Tehát szerintünk a három különböző program a megfelelő tervezés.

6-6. feladat. Értelmesen működik-e a **p**, ha a **pagesize** nem pozitív? □

6-7. feladat. Mit lehetne még csinálni a **p**-vel? Módosítsuk úgy, hogy a korábbi bemenetrészeket újra ki tudja írni! (Ez egy élvezetes különleges tulajdonság.) Lehessen egy egységben a teljes képernyőnél kevesebbet is kiírni. Bővítsük olyan lehetőséggel, hogy előre vagy hátra tudjon keresni egy számával vagy tartalmával megadott sort! □

6-8. feladat. Használjuk fel a burok **exec** beépített függvényének (l. **sh(1)**) fájlkezelő lehetőségeit arra, hogy a **system** számára rögzítsük a **ttyin** hívását! □

6-9. feladat. Ha elfelejtünk bemenetet adni a **p**-nek, akkor csendben áll és várja a terminálról érkező bemenetet. Érdemes-e jelezni ezt a lehetséges hibát? Ha igen, hogyan? Útmutatás: l. **isatty(3)**! □

6.5. Egy példa: pick

A **pick** 5. fejezetben szereplő változata kiterjesztette a burok lehetőségeit. Az itt következő C nyelvű változat kicsit eltér az 5. fejezetbelitől. Ha vannak argumentumai, akkor ugyanúgy dolgozza fel azokat, mint korábban. De ha csak egyetlen mínuszjel (−) argumentumot adunk meg, akkor a **pick** a szabványos bemenetet dolgozza fel.

Ha nincs argumentum, akkor miért nem csak beolvassa a szabványos bemenetet? Tekintsük a **zap** parancs 5.6. szakaszban szereplő második változatát:

```
kill $SIG `pick \`ps -ag | egrep "$*"` | awk '{print $1}'`
```

Mi történik, ha az **egrep** minta nem egyezik semmivel? Ebben az esetben a **pick**-nek nincs argumentuma és szabványos bemenetét kezdi olvasni, a **zap** parancs érthetetlen módon hibázik. Egy explicit argumentumot megkövetelve egyszerűen elkerülhetjük az ilyen helyzetet. A **cat**-ból és más programokból származó mínuszjelre vonatkozó megállapodás mutatja, hogyan adjuk meg az argumentumot.

```
/* pick: minden argumentumra választást ajánl */

#include <stdio.h>
char *progname;      /* programnév hibaüzenetekben */

main(argc, argv);
  int argc;
  char *argv[];
{
  int i;
  char buf[BUFSIZ];

  progname = argv[0];
  if (argc == 2 && strcmp(argv[1], "-") == 0) /* - kiválasztás */
    while (fgets(buf, sizeof buf, stdin) != NULL) {
      buf[strlen(buf)-1] = '\0' ; /* újsor eldobása */
      pick(buf);
    }
  else
    for (i = 1; i <argc; i++)
      pick(argv[i]);
  exit (0);
}
pick(s)   /* s választást ajánl */
  char *s;
{
  fprintf(stderr, "%s? ", s);
  if (ttyin() == 'y')
    printf("%s\n", s);
}
```

A **pick** egy programban megvalósítja az argumentumok interaktív válogatásának lehetőségét. Ez a hasznos szolgáltatás más parancsoknál csökkentheti az „interaktív" opciók szükségességét.

6-10. feladat. Ha adott a **pick**, kell-e az **rm −i** parancs? □

6.6. Hibák és nyomkeresés

Aki valaha írt már programot, annak ismerős a hiba fogalma. Nincs recept hibátlan program írására, csak annyit tehetünk, hogy áttekinthető, egyszerű szerkezetet tervezünk, gondosan valósítjuk meg, és módosításkor is megtartjuk áttekinthetőségét.

Van néhány UNIX eszköz, amely segít megtalálni a hibákat, bár egyik sem elsőosztályú. Ezek bemutatásához szükségünk van egy hibára. Könyvünkben azonban az összes program tökéletes! Mit tehetünk? Szándékosan létrehoztunk egy tipikus hibát. Tekintsük a fenti **pick** függvényt! Itt van újra, de most van benne egy hiba (ne nézzük meg újra ez eredetit!):

```
pick(s) /* s választást ajánl */
  char *s;
{
fprintf("%s? ", s);
  if (ttyin() == 'y')
    printf("%s\n", s);
}
```

Mi történik, ha lefordítjuk és lefuttatjuk?

```
$ cc pick.c -o pick
$ pick *.c                          Próbáljuk ki!
Memory fault - core dumped          Katasztrófa!
$
```

A Memory fault hibaüzenet azt jelenti, hogy programunk egy olyan tárterületre hivatkozott, amelyre nem szabad. Ez általában azt jelenti, hogy egy mutató valamilyen vad helyre mutat. A Bus error egy másik hasonló jelentésű hibaüzenet, amelyet gyakran egy nem lezárt karakterlánc végignézése okoz.

A **Core dumped** azt jelenti, hogy a mag kimentette futó programunk állapotát munkajegyzékünkbe, egy **core** nevű fájlba. Az előtérben futó programról *ctl-* beütésével, a háttérben futóról a **kill −3** paranccsal kaphatunk kiírást a tártartalomról.

Két program van, amelyekkel nézelődhetünk: az **adb** és az **sdb.** Mint a legtöbb nyomkereső program, ezek is misztikusak, bonyolultak és nélkülözhetetlenek. Az **adb** megtalálható UNIX 7. kiadásában, az **sdb** csak a rendszer későbbi változataiban, de az egyik mindig megtalálható.

Itt csak legelemibb használatukat mutathatjuk be: veremnyom-kiíratás, amikor kiírják a program halála előtt futott utolsó függvényt, az azt hívó függvényt és így tovább. A veremnyomban levő első megnevezett függvény az, amelyben a program elszáll.

Az **adb** a veremnyomot a $C paranccsal adja:

```
$ adb pick core                                  Hívja az adb-t!
$C                                               Veremnyomkérés.
~ _strout(0175722,011,0,011200)
        adjust:         0
        fillch:         060542
_doprnt(0177345,0176176,011200)
~fprintf(011200,0177345)
        iop:            011200
        fmt:            0177345
        args:           0
~pick(0177345)
        s:              0177345
~main(035,0177234)
        argc:           035
        argv:           0177234
        i:              01
        buf:            0
ctl-d                                            Kilép.
$
```

Ez azt mondja, hogy a **main** hívta a **pick**-et, ami hívta az **fprintf**-et, ami hívta a **_doprnt**-t, ami hívta a **_strout**-ot. Miután **_doprnt**-t a **pick.c**-ben sehol sem említjük, a probléma valahol az **fprintf**-ben vagy fölötte lehet. (A veremnyomban a szubrutinok utáni sorok a helyi változók értékét mutatják. A *$c* elhagyja ezt az információt, az **adb** néhány változatában *$C* is ezt teszi.)

Mielőtt mindent felfedeznénk, próbáljuk meg ugyanezt az **sdb**-vel:

```
$ sdb pick core
Warning:`a.out' not compiled with -g
lseek: address 0xa64                             Rutin, amelyben a program meghalt.
*t                                               Veremnyomkérés.
lseek()
fprintf(6154,2147479154)
pick(2147479154)
main(30,2147478988,2147479112)
*q                                               Kilép.
$
```

Az információt másképp csoportosítja, mint az **adb,** de van egy közös bennük: az **fprintf.** (A kiíratás különböző, mert ez egy másik gépen futott – egy VAX-11/750 gépen –, amely másképp valósítja meg a szabványos be- és kimeneti könyvtárat. Hogy bizonyságot kapjunk, nézzük meg a **pick** hibás változatában az **fprintf**-hívást; látjuk, hogy az rossz:

```
fprintf("%s? ", s);
```

Kimaradt az **stderr**, így a "%s? " formátum-karakterláncot **FILE** mutatóként használja, és természetesen ennek káosz lesz a következménye.

Azért választottuk ezt a hibát, mert gyakori, és inkább elnézés, mint rossz tervezés következménye. Az ilyen hibákat, amelyekben a függvényt rossz argumentummal hívjuk be, meg lehet találni a C nyelvű programokat ellenőrző **lint(1)** programmal. A **lint** a C nyelvű programokat megvizsgálja lehetséges hibák, hordozhatóság és kétes szerkezetek szempontjából. Ha a **lint**-et a teljes **pick.c** fájlra lefuttatjuk, akkor megjelenik a hiba:

```
$ lint pick.c
...

fprintf, arg. 1 used inconsistently "llib-lc"(69) :: "pick.c"(28)

...
$
```

ami azt jelenti, hogy az **fprintf** első argumentuma a szabványos könyvtári definíciójában eltér a programunk 28. sorában használttól. Ez a hiba megtalálásához komoly útmutatás.

A **lint** egy vegyes felvágott. Pontosan megmondja, hogy mi a hiba a programban, de sok lényegtelen üzenetet is közöl, amit példánkban elhagytunk, mert némi tapasztalat kell ahhoz, hogy tudjuk, mire figyeljünk, és mire ne. A **lint** használata megéri a fáradtságot, mert olyan hibákat is megtalál, amelyeket az ember szinte sosem vesz észre. Egy hosszabb szerkesztési rész után általában már érdemes lefuttatni a **lint**-et, és minden figyelmeztetését ellenőrizni.

6.7. Egy példa: zap

A **zap** is az 5. fejezetben burokfájlként bemutatott program, amely válogatva töröl folyamatokat. A burokváltozat fő hibája a sebessége: olyan sok folyamatot hoz létre, hogy lassan fut. Ez különösen nem kívánatos kóbor folyamatokat törlő programnál. A **zap** C nyelvre átírva gyorsabbá válik. Nem az egészet írjuk át, továbbra is a **ps** alkalmazásánál fogjuk a folyamatinformációt keresni. Ez *sokkal egyszerűbb,* mint a magból kiásni az információt, és ez is hordozható. A **zap** bemenete egy **ps**-sel megnyitott cső, és fájl helyett innen olvas. A **popen(3)** függvény analóg az **fopen**-nel, de első argumentuma nem fájlnév, hanem egy parancs. Van egy **pclose** is, de nekünk most nem kell.

```
/* zap: interaktív folyamat törlő */

#include <stdio.h>
#include <signal.h>
char *progname; /* programnév hibaüzenetekhez */
char *ps = "ps -ag"; /* rendszerfüggő */
```

```
main(argc, argv)
  int argc;
  char *argv[];
{
  FILE *fin, *popen();
  char buf[BUFSIZ];
  int pid;

  progname = argv[0];
  if ((fin = popen(ps, "r")) == NULL) {
    fprintf(stderr, "%s: can't run %s\n", progname, ps);
    exit(1);
  }
  fgets(buf, sizeof buf, fin); /* veszi a fejlécsort */
  fprintf(stderr, "%s", buf);
  while (fgets(buf, sizeof buf, fin) != NULL)
    if (argc == 1 || strindex(buf, argv[1]) >= 0) {
      buf[strlen(buf)-1] = '\0'; /* \n elnyomása */
      fprintf(stderr, "%s? ", buf);
      if (ttyin() == 'y') {
        sscanf(buf, "%"d, &pid);
        kill(pid, SIGKILL);
      }
    }
  exit(0);
}
```

Programunkba beépítettük ugyan a **ps −ag**-t (az opció rendszerfüggő), de ha nem vagyunk szuperfelhasználók, akkor csak a saját folyamatainkat tudjuk törölni.

Az első **fgets** hívás a **ps** fejlécsorát veszi; egy érdekes feladat annak kikövetkeztetése, hogy mi lesz, ha megpróbáljuk a fejlécsornak megfelelő „folyamatot" törölni.

Az **sscanf** függvény az **scanf(3)** család tagja, bemenetiformátum-átalakítást végez. Nem fájlból, hanem karakterláncból konvertál. A **kill** rendszerhívás meghatározott jelet küld a folyamatnak; a **<signal.h>**-ban definiált **SIGKILL** jelet nem lehet elfogni vagy törölni. Emlékezhetünk az 5. fejezetből, hogy **SIGKILL** számértéke 9, de jobb gyakorlat a fejlécfájlok szimbolikus állandóit használni, mint hűvös számokkal teletűzdelni a programot.

Ha nincs argumentum, akkor a **zap** minden **ps** kimenő sort átad válogatásra. Ha van argumentum, akkor a **zap** csak azokat a **ps** kimenő sorokat adja át, amelyek a mintával egyeznek. Az **strindex(s1, s2)** függvény ellenőrzi, hogy az argumentum egyezik-e a **ps** kimenet sorának bármely részével. Ha az **strncmp**-t használjuk (l. a 6.2. táblázatot), akkor az **strindex** vagy azt az **s1** pozíciót adja vissza, amelyben az **s2** előfordult, vagy ha nem talált **s2**-t, akkor értéke −1.

```
strindex(s, t) /* visszaadja t s-beli indexét, ha nincs, −1-et */
  char *s, *t;
```

```
{
  int i, n;
  n = strlen(t);
  for (i = 0; s[i] != '\0'; i++)
    if (strncmp(s+i, t, n) == 0)
      return i;
  return -1;
}
```

A 6.4. táblázat összefoglalja a szabványos be- és kimeneti könyvtár leggyakrabban használt függvényeit.

6-11. feladat. Módosítsuk úgy a **zap**-ot, hogy bármilyen számú argumentumot megadhassunk neki! A példánkbeli **zap** egyik választása lehetőségként normál esetben a

6.4. táblázat

Hasznos szabványos be- és kimeneti függvények

fp = fopen(s,mód)	megnyitja az s fájlt; a mód lehet "r", "w", "a" (olvasás, írás, hozzáfűzés), NULL-t ad vissza, ha hiba volt
c = getc(fp)	vesz egy karaktert; **getchar( )** ugyanaz, mint **getc(stdin)**
putc(c,fp)	kivisz egy karaktert; **putchar(c)** ugyanaz, mint **putc(c,stdout)**
ungetc(c,fp)	visszateszi a karaktert az **fp** bemeneti fájlba, egyszerre legfeljebb 1 karakter tolható vissza
scanf(fmt,a1,...)	**stdin**-ből a1,...-be **fmt** szerint karaktereket olvas. Mindegyik a_i csak mutató lehet. EOF-ot vagy az átalakított mezők számát adja vissza
fscanf(fp,...)	az **fg** fájlból olvas
sscanf(s,...)	az **s** karakterláncból olvas
printf(fmt,a1,...)	a1,...-et az **fmt** szerint formázva kiírja a **stdout**-ra
fprintf(fp,...)	kiírja a ...-ot az **fp** fájlba
sprintf(s,...)	kiírja a ...-ot az **s** karakterláncba
fgets(s,n,fp)	**fp**-ből **s**-be legfeljebb n karaktert beolvas. NULL-t ad vissza, ha fájl vége van
fputs(s,fp)	az **s** karakterláncot az **fp** fájlba írja
fflush(fp)	minden pufferezett kimenetet kiürít az **fp** fájlba
fclose(fp)	lezárja az **fp** fájlt
fp = popen(s,mód)	az **s** parancs számára csövet nyit meg. L. **fopen.**
pclose(fp)	lezárja az fp csövet
system(s)	futtatja az **s** parancsot és várja annak befejeződését

saját magának megfelelő sort visszhangozza. Jó-e ez így? Ha nem, módosítsuk a programot! Útmutatás: l. **getpid(e).** □

6-12- feladat. Építsünk egy **fgrep(1)**-et a **strindex** köré! Hasonlítsuk össze a futási időket, mondjuk egy nagyobb anyagban 10 szó kikeresésénél! Miért fut az **fgrep** gyorsabban? □

6.8. Egy interaktív fájl-összehasonlító program: idiff

Gyakran előfordul, hogy egy fájlból két, kissé eltérő változatunk van, és mindkettőben vannak nélkülözhetetlen részek. Ez a helyzet sokszor előáll, amikor két ember egymástól függetlenül változtat egy fájlon. A **diff** megmondja, hogy a fájlok miben különböznek, de közvetlenül nem segít a részek összeválogatásában.

Ebben a szakaszban írunk egy **idiff** (interaktív **diff**) nevű programot, amely a **diff**-kimenet minden elemét előállítja és lehetővé teszi a felhasználónak a „tól"-rész, az „ig"-rész vagy a szerkesztendő rész kiválasztását. Az **idiff** a kiválasztott részeket megfelelő sorrendbe állítja és az **idiff.out** fájlba helyezi. Ha adott két fájl:

fájl1:
```
Ez itt
vizsgálja
a
te
ügyességedet
és tehetségedet.
```

fájl2:
```
Ez itt
nem vizsgálja
a
mi
képességeinket.
```

akkor a **diff** a következőt adja:

```
$ diff file1 file2
2c2
< vizsgálja
---
> nem vizsgálja
4,6c4,5
< te
< ügyességedet
< és tehetségedet.
---
> mi
> képességeinket.
$
```

Az **idiff**-fel a párbeszéd a következő lehet:

```
$ idiff file1 file2
2c2                                    Az első eltérés.
```

```
< vizsgálja
---
> nem vizsgálja
? >                                   A felhasználó a második (>) változatot választja.
4,6c4,5
< te                                  A második eltérés.
< ügyességedet
< és tehetségedet.
---
> mi
> képességeinknek.
? <                                   A felhasználó az első (<) változatot választja.
idiff output in file idiff.out
$ cat idiff.out                       A kimenetet ebbe a fájlba tette.
Ez itt
nem vizsgálja
a
te
ügyességedet
és tehetségedet.
$
```

Ha a < vagy > jelek helyett az *e* választ adjuk, akkor az **idiff** a két, már bennlevő sorcsoportra hívja az **ed**-et. Ha pl. a második válasz *e* lett volna, akkor a szerkesztőprogram pufferébe a következő került volna:

```
te
ügyességedet
és tehetségedet.
---
mi
képességeinket.
```

Amit az **ed**-del visszaírunk a fájlba, az lesz a végső kimenet.

Végül az **idiff**-ből a **!cmd**-vel kiszökve bármilyen parancsot végrehajthatunk.

Technikailag a munka nehezebb része a **diff** lenne, de az készen áll számunkra. Így feladatunk valójában a **diff** kimenetének elemzése, és a megfelelő fájlok megfelelő időben való megnyitása, lezárása, olvasása, írása. Az **idiff** főrutinja felépíti a fájlokat és futtatja a **diff** folyamatot.

```
/* idiff: interaktív diff */

#include <stdio.h>
#include <ctype.h>
char *progname;
#define HUGE 10000                    /* nagyon sok fájl */
main(argc, argv)
  int argc;
  char *argv[];
```

```
{
  FILE *fin, *fout, *f1, *f2, *efopen();
  char buf[BUFSIZ], *mktemp();
  char *diffout = "idiff.XXXXXX";
  progname = argv[0];
  if (argc != 3) {
    fprintf(stderr, "Usage: idiff file1 file2\n");
    exit(1);
  }
  f1 = efopen(argv[1], "r");
  f2 = efopen(argv[2], "r");
  fout = efopen("idiff.out", "w");
  mktemp(diffout);
  sprintf(buf, "diff %s %s >%s", argv[1], argv[2], diffout);
  system(buf);
  fin = efopen(diffout, "r");
  idiff(f1, f2, fin, fout);
  unlink(diffout);
  printf("%s output in file idiff.out\n", progname);
  exit(0);
}
```

Az **mktemp(3)** függvény olyan fájlt hoz létre, amelynek neve garantáltan különbözik az összes létező fájl nevétől. Az **mktemp** az argumentumba a hat *X* helyére az **idiff** folyamatazonosítóját és egy betűt ír. Az **unlink(2)** rendszerhívás a megnevezett fájlt törli a fájlrendszerből.

A **diff** által kijelzett eltérések végigpásztázását az **idiff** nevű függvény végzi. Az alapötlet elég egyszerű: kiír egy egységet a **diff** kimenetéből az egyik fájlba, átugorja a nemkívánt részt, majd egy másikból bemásolja a kívánt változatot. Sok unalmas részlet van benne, úgyhogy a programlista nagyobb, mint szerettük volna, de elég egyszerű ahhoz, hogy részenként meg lehessen érteni.

```
idiff(f1, f2, fin, fout) /* eltérések feldolgozása */
  FILE *f1, *f2, *fin, *fout;
{
  char *tempfile = "idiff.XXXXXX";
  char buf[BUFSIZ], buf2[BUFSIZ], *mktemp();
  FILE *ft, *efopen();
  int cmd, n, from1, to1, from2, to2, nf1, nf2;
  mktemp(tempfile);
  nf1 = nf2 = 0;
  while (fgets(buf, sizeof buf, fin) != NULL) {
    parse(buf, &from1, &to1, &cmd, &from2, &to2);
    n = to1-from1 + to2-from2 + 1; /* sorok száma eltérő */
    if (cmd == 'c')
      n += 2;
    else if (cmd == 'a')
      from1++;
    else if (cmd == 'd')
      from2++;
```

```
    printf("%s", buf);
    while (n-- > 0) {
      fgets(buf, sizeof buf, fin);
      printf("%s", buf);
    }
    do {
      printf("? ");
      fflush(stdout);
      fgets(buf, sizeof buf, stdin);
      switch (buf[0]) {
      case '>':
        nskip(f1, to1-nf1);
        ncopy(f2, to2-nf2, fout);
        break;
      case '<':
        nskip(f2, to2-nf2);
        ncopy(f1, to1-nf1, fout);
        break;
      case 'e':
        ncopy(f1, from1-1-nf1, fout);
        nskip(f2, from2-1-nf2);
        ft = efopen(tempfile, "w");
        ncopy(f1, to1+1-from1, ft);
        fprintf(ft, "---\n");
        ncopy(f2, to2+1-from2, ft);
        fclose(ft);
        sprintf(buf2, "ed %s", tempfile);
        system(buf2);
        ft = efopen(tempfile, "r");
        ncopy(ft, HUGE, fout);
        fclose(ft);
        break;
      case '!':
        system(buf+1);
        printf("!\n");
        break;
      default:
        printf("< or > or e or !\n");
        break;
      }
    } while (buf[0]!='<' && buf[0]!='>' && buf[0]!='e');
    nf1 = to1;
    nf2 = to2;
  }
  ncopy(f1, HUGE, fout); /* nagyon nagy fájloknál hibás lehet */
  unlink(tempfile);
}
```

A **parse** függvény végzi a **diff** által előállított sorok elemzésének prózai, de trükkösen megoldott feladatát. Ehhez a négy sorszámot és a parancsot (**a**, **c** vagy **d** valamelyikét) használja. A **parse** egy kicsit bonyolult program, mert a **diff** a parancs betűjének mindkét oldalán egy vagy két sorszámot is adhat.

```
parse(s, pfrom1, pto1, pcmd, pfrom2, pto2)
  char *s;
  int *pcmd, *pfrom1, *pto1, *pfrom2, *pto2;
{
#define a2i(p) while (isdigit(*s)) p = 10*(p) + *s++ - '0'
  *pfrom1 = *pto1 = *pfrom2 = *pto2 = 0;
  a2i(*pfrom1);
  if (*s == ',') {
    s++;
    a2i(*pto1);
  } else
    *pto1 = *pfrom1;
  *pcmd = *s++;
  a2i(*pfrom2);
  if (*s == ',') {
    s++;
    a2i(*pto2);
  } else
    *pto2 = *pfrom2;
}
```

Az **a2i** makró végzi az itt szükséges speciális ASCII – egész szám konverziót a négy előfordulási helyen.

Az **nskip** és **ncopy** a fájlból megadott számú sort átugrik, ill. átmásol:

```
nskip(fin, n) /* n sor átugrása az fin fájlban* /
  FILE *fin;
{
  char buf[BUFSIZ];
  while (n-- > 0)
    fgets(buf, sizeof buf, fin);
}
ncopy(fin, n, fout) /* a fin fájlból n sort az fout fájlba másol */
  FILE *fin, *fout;
{
  char buf[BUFSIZ];
  while (n-- > 0) {
    if (fgets(buf, sizeof buf, fin) == NULL)
      return;
    fputs(buf, fout);
  }
}
```

Láthatjuk, hogy az **idiff** nem valami kecsesen lép ki, ha megszakítjuk, hiszen számos fájlt hagy a **/tmp**-ben. A következő fejezetben megmutatjuk, hogyan lehet az ideiglenes fájlok törléséhez a megszakítást elfogni.

Mind a **zap,** mind az **idiff** esetében a leglényegesebb megfigyelés az, hogy a nehéz munka nagy részét valaki más már elvégezte. Ezek a programok valójában csak egy kényelmes kapcsolatot teremtenek egy másik, a kívánt információt adó programmal. Érdemes keresni az ilyen lehetőségeket, más munkájára építeni ahelyett, hogy mi magunk csinálnánk az egészet – egyszerű módja ez annak, hogy termelékenyebbek legyünk.

6-13. feladat. Bővítsük az **idiff**-et egy **q** paranccsal: a *q*< válasz jelentse azt, hogy a hátralevő rész minden választása automatikusan < válasz; a *q*> pedig azt, hogy > válasz! □

6-14. feladat. Módosítsuk az **idiff**-et úgy, hogy bármelyik **diff** argumentumot továbbadja a **diff**-nek! (**−b** és **−h** a legesélyesebb jelöltek.) Módosítsuk az **idiff**-et úgy, hogy másik szerkesztőt lehessen megadni neki, így:

```
$ idiff −e másik_szerkesztő fájl1 fájl2
```

Hogyan hat egymásra ez a két módosítás? □

6-15. feladat. Változtassuk meg az **idiff**-et úgy, hogy a **diff** kimenetének ideiglenes fájlja helyett a **popen**-t és a **pclose**-t használja! Milyen változást okoz ez a program sebességében és bonyolultságában? □

6-16. feladat. A **diff**-nek van egy tulajdonsága: ha egyik argumentuma egy jegyzék, akkor abban keres egy, a másik argumentummal egyező nevű fájlt. Ha ugyanezt megpróbáljuk az **idiff**-fel, akkor különös módon hibát kapunk. Magyarázzuk meg, mi történik, azután javítsuk ki a hibát! □

6.9. A környezet elérése

A burok környezeti változóihoz C nyelvű programból könnyű hozzáférni. Ennek alapján néha a programokat úgy illeszthetjük a környezetünkbe, hogy ne követeljünk sokat felhasználóiktól. Pl. tegyük fel, hogy olyan terminált használunk, amelynek képernyőmérete nagyobb, mint a normál 24 sor. Ha a **p**-t használjuk és terminálunk összes előnyét szeretnénk élvezni, milyen választási lehetőségek állnak előttünk? Kellemetlen, ha minden **p** híváskor definiálnunk kell a képernyő méretét:

```
$ p −36 ...
```

Tehetünk egy burokfájlt a **bin** jegyzékünkbe:

```
$ cat /usr/you/bin/p
exec /usr/bin/p -36 $*
$
```

Harmadik megoldás lehet, hogy a **p** programba beépítünk egy, a terminál tulajdonságait meghatározó környezeti változót. Tegyük fel, hogy a **PAGESIZE** változót **.profile**-ban definiáljuk:

```
PAGESIZE=36
export PAGESIZE
```

A **getenv("var")** rutin végigkeresi a környezetet a **var** burokváltozóért, és annak értékét karakterlánc formájában visszaadja, ill. **NULL**-t ad, ha a változó nincs definiálva. A **getenv** felhasználásával könnyű a **p**-t módosítani. Annyit kell csak tennünk, hogy néhány deklarációt hozzáírunk és a főrutin elején hívjuk a **getenv**-et:

```
/* p: egységenként kiírja a bemenetet (3. változat) */
...
  char *p, *getenv();
  progname = argv[0];
  if ((p=getenv("PAGESIZE")) != NULL)
    pagesize = atoi(p);
  if (argc >1 && argv[1][0] == '-') {
    pagesize = atoi(& argv[1][1]);
    argc--;
    argv++;
  }
...
```

Az opcionális argumentumok a környezeti változó után kerülnek feldolgozásra, így bármilyen közvetlenül megadott oldalméret mindig felülírja a közvetett módon megadott méretet.

6-17. feladat. Módosítsuk az **idiff**-et úgy, hogy keresse végig a környezetet az alkalmazandó szerkesztőprogram nevéért! Módosítsuk a **2, 3** stb. programokat úgy, hogy használják a **PAGESIZE**-t! □

Történeti és irodalmi megjegyzések

A szabványos be- és kimeneti könyvtárat Mike Lesk hordozható könyvtára alapján Dennis Ritchie tervezte. Mindkét programcsomagnak alapvetően elegendő szabványos lehetőséget kívánt biztosítani ahhoz, hogy a programokat UNIX rendszerből nem UNIX rendszerekbe változtatás nélkül át lehessen vinni.

A mi **p** programunk terve Henry Spencer egy programján alapszik.

Az **adb** szerzője Steve Bourne, az **sdb** készítője Howard Katseff, a **lint** pedig Steve Johnson alkotása.

Az **idiff** egy Joe Maranzano által írt program alapján készült.

Maga a **diff** Doug McIlroy műve, amely egy Harold Stone, Wayne Hunt és Tom Szymanski által egymástól függetlenül kitalált algoritmuson alapszik. (L. J. W. Hunt és T. G. Szymanski „A fast algorithm for computing longest common sequences", CACM. 1977. május). A **diff** algoritmust M. D. McIlroy és J. W. Hunt írták le „An algorithm for differential file comparison" című munkájukban (Bell Labs Computing Science Technical Report 41, 1976.). A **diff** program fejlődéséről McIlroy-t idézzük: „Legalább három teljesen különböző algoritmust próbáltam ki, mielőtt a végsőhöz jutottam. A **diff** nagyszerűen mutatja, hogy nem szabad egy programban mereven ragaszkodni valamihez, hanem addig kell a programot változtatni, amíg jó lesz".

7. Rendszerhívások

Ez a fejezet a UNIX operációs rendszerrel való kapcsolat legalsó szintjével, a rendszerhívásokkal, azaz a mag belépési pontjaival foglalkozik. Az operációs rendszer csupán azokat a lehetőségeket nyitja, amelyek a rendszerhívásokkal érhetők el; minden más ezekre épül.

A rendszerhívásokat több nagy csoportra osztjuk. Az első a be- és kiviteli rendszer, az **fopen** és **putc** rutinokkal rokon alapvető könyvtári rutinok csoportja. Bővebben fogjuk tárgyalni a fájlrendszert is, különösen a jegyzékeket és az i-bögöket. Ezután az eljárások csoportja következik, amelyből megtudjuk, hogyan kell programból programokat futtatni. Végül a jelek és a megszakítások csoportját ismertetjük, amelyből kiderül pl. hogy mi történik, ha lenyomjuk a *DELETE* billentyűt, és hogyan lehet ezt értelmesen kezelni a programban.

Példáink többsége – akárcsak a 6. fejezetben olyan hasznos program, amely a UNIX 7. kiadásában még nem szerepelt. Átnézésükből még azok is tanulhatnak valamit, akik nem közvetlenül ezeket a programokat akarják hasznosítani. Számukra a példák átnézése sugalmazhatja, hogy rendszerükbe hasonló eszközöket építsenek.

A rendszerhívások teljes leírása a UNIX kézikönyv 2. fejezetében található. Mi ebben a fejezetben a legfontosabb részekről beszélünk, de nem törekszünk a teljességre.

7.1. Alacsony szintű be- és kimenet

A legegyszerűbb be- és kimeneti műveletek közvetlen belépések az operációs rendszerbe. A programok fájlokat írnak vagy olvasnak alkalmas méretű darabokban. A darabokat a külső egységeknek megfelelő méretű pufferekbe pakolja, és úgy ütemezi az egységek tevékenységét, hogy hatékonyságuk az összes felhasználót tekintve optimális legyen.

Fájlleírók

Az összes be- és kimenet fájlok olvasását, ill. írását jelenti, mivel minden perifériaegység, még a képernyő is a fájlrendszer egy fájlja. Ez azt jelenti, hogy a program és minden perifériaegység közötti adatátvitel egyetlen egy szabvány szerint folyik.

Általában egy fájl írása vagy olvasása előtt közölnünk kell az operációs rendszerrel szándékunkat. Ezt a tevékenységet a fájl *megnyitásának* nevezzük. (Ha írni akarunk, akkor lehet, hogy *létre kell hozni* a fájlt.) Az operációs rendszer ellenőrzi, hogy szándékunkhoz van-e jogunk. (Létezik a fájl? Van-e engedélyünk az elérésére?) Ha minden rendben van, akkor a megnyitás után kapunk egy nemnegatív egész számot. Ezt a számot nevezzük *fájlleírónak*. Az összes írási vagy olvasási műveletben ténylegesen a fájlleírót használjuk, és nem a fájl nevét. A megnyitott fájlra vonatkozó összes információt a rendszer kezeli, programunk a fájlra csak a fájlleíróval hivatkozik. A 6. fejezetben tárgyalt **FILE** egy olyan adatszerkezetre mutat, amely többek között a fájlleírót tartalmazza; a **<stdio.h>**-ban definiált **fileno(fp)** makró a fájlleírót adja vissza.

A terminálra való írást és onnan való olvasást különleges eljárások könnyítik meg. Amikor egy programot a burok elindít, akkor az három megnyitott fájlt örököl; a szabványos bemenet, a szabványos kimenet és a szabványos hibafájlokat, amelyek leírói: **0, 1** és **2**. Mindezek alapértelmezésben a terminálhoz tartoznak, tehát ha egy program csak a **0**-ás fájlleíróval rendelkező fájlból olvas és az **1**-es, ill. **2**-es fájlleírójú fájlokba ír, akkor fájlmegnyitás nélkül írhat és olvashat. Ha a program más fájlokat is megnyit, akkor azok sorra a **3**-as és **4**-es stb. fájlleírókat kapják.

Ha a be- és kimenetet fájlokról vagy csövekről átirányítjuk, a burok a **0**-ás és **1**-es fájlleírók alapértelmezés szerinti hozzárendelését a terminálról a megnevezett fájlra változtatja. Általában a **2**-es fájlleíró továbbra is a terminálhoz tartozik, így a hibaüzenetek ott jelennek meg. Az olyan bűvös parancsok, mint **2>fájlnév** vagy **2>&1**, az alapértelmezés megváltoztatására vezetnek, de a fájlhozzárendeléseket nem a program, hanem a burok változtatja meg. (A program maga is megváltoztathatja ezt a későbbiekben, ha akarja, de ez ritkaságszámba megy.)

Fájlbemenet és -kimenet: read és write

Az összes be- és kimenetet két operációsrendszer-hívás, a **read** és a **write** hajtja végre, amelyek C nyelvű programból ugyanilyen nevű függvényként érhetők el. Mindkét függvény első argumentuma a fájlleíró. A második argumentum egy bájtokból álló tömb, amely az adatok forrásaként vagy céljaként szolgál. A harmadik argumentum az átviendő bájtok száma.

```
int fd, n, nread, nwritten;
char buf[SIZE];
nread = read(fd, buf, n);
nwritten = write(fd, buf, n);
```

Mindegyik hívás visszaadja az átvitt bájtok számát. Olvasáskor ez az érték kisebb is lehet, mint amennyit kértünk, hiszen előfordulhat, hogy n-nél kevesebb bájtnyi információ várt csak beolvasásra. (Amikor a fájl terminálról érkezik, akkor a **read** általában csak a következő újsor-karakterig olvas, tehát többnyire kevesebbet, mint amennyit kértünk.) A művelet befejezése után kapott 0 érték a fájlvéget, −1 valami-

lyen hibát jelez. Íráskor a visszaadott érték a ténylegesen kiírt bájtok száma, ha ez nem egyenlő a kiírandó bájtok számával, akkor a művelet hibás.

Bár a bájtok számát semmi sem korlátozza, a két leggyakoribb érték az 1, ami a karakterenkénti (pufferelés nélküli) műveletet jelenti, ill. a lemezblokk mérete, ami általában 512 vagy 1024. (Az **<stdio.h>**-ban szereplő **BUFSIZ** paraméternek ez az értéke.)

Példaként tekintsünk egy olyan programot, amely a bemenetét a kimenetére másolja. Mivel a bemenet és a kimenet bármely fájlra vagy egységre átirányítható, programunk bármit bárhova másol: egyszerű megvalósítása ez a **cat** programnak.

```
/* cat: legegyszerűbb változat */
#define SIZE 512 /* tetszőleges érték */
main()
{
    char buf[SIZE];
    int n;
    while ((n = read(0, buf, sizeof buf)) > 0)
        write(1, buf, n);
    exit(0);
}
```

Ha a fájl mérete nem a **SIZE** többszöröse, akkor az egyik **read** ennél kisebb értéket fog a **write** számára visszaadni; azután a következő **read** hívás már 0-t ad vissza.

A lemezblokkoknak megfelelő méretben olvasni és írni gyorsabb, mint a karakterenkénti, de kisebb adatmennyiségnél elfogadható a karakterenkénti is. Mivel a mag pufferei adatainkat, számunkra a fő terhet a rendszerhívások jelentik. Az **ed** pl. karakterenként olvassa a szabványos bemenetet. Megmértük a **cat** program előbbi változatának futási idejét egy 54000 bájtos fájlon, hatféle **SIZE** esetén, az eredmény:

	Idő (program + rendszeridő, s)	
SIZE	PDP−11/70	VAX−11/750
1	271,0	188,8
10	29,9	19,3
100	3,8	2,6
512	1,3	1,0
1024	1,2	0,6
5120	1,0	0,6

A lemezblokk mérete a PDP−11-en 512 bájt és a VAX-on 1024.

Különböző folyamatok használhatják egyazon időben ugyanazt a fájlt, sőt, az egyik folyamat írhat, amíg a másik olvas. Ha ezt nem akarjuk, akkor letilthatjuk, de néha érdemes megengedni. Közös fájlhasználat során előfordulhat, hogy egy **read** 0-val tér vissza, azaz fájl végét jelez, de a következő **read** már adatokat fog elérni! A **readslow** program ezt figyelembe véve a fájl vége jelzéstől függetlenül folytatja a beolvasást. A **readslow** program jó eszköz arra, hogy egy program működésének folyamatát figyelemmel kísérjük.

```
$ slowprog >múlandó &
5213                          Folyamatazonosító.
$ readslow <múlandó | grep valami
```

Más szóval egy lassú program egy fájlba beírja a kimenetét, míg a **readslow** program – esetleg más programokkal együtt – figyeli a felgyülemlő adatokat.

A **readslow** program azonos a **cat** programmal, de nem lép ki a ciklusból, ha a bemeneten fájl véget érzékel. Rendszerhívásokat kell használnia, mivel a szabványos könyvtári be- és kimeneti rutinok az első fájl vége érzékelése után mindig **EOF**-ot adnak vissza.

```
/* readslow: folyamatosan olvas, vár */
#define SIZE 512 /* tetszés szerinti */
main()
{
  char buf [SIZE];
  int n;
  for (; ;) {
    while ((n = read(0, buf, sizeof buf)) > 0)
      write(1, buf, n);
    sleep(10);
  }
}
```

A **sleep** függvény a paraméterként megadott másodpercig felfüggeszti a programot (leírása a sleep(3)-ban található). Nem akarjuk, hogy a **readslow** állandóan a fájl körül szöszmötöljön újabb adatokra várva, mivel ez túl sok CPU-időbe kerülne. A **readslow** ezen változata a fájl végéig másolja a bemenetet, alszik egyet, majd újra próbálkozik. Ha újabb adatok érkeztek, amíg aludt, akkor azokat a következő **read**-del be fogja olvasni.

7-1. feladat. Egészítsük ki **readslow**-t egy $-n$ argumentummal, hogy az alapértelmezés szerinti várakozási időt meg lehessen változtatni n másodpercre! Egyes UNIX rendszerekben a **tail**-nek van egy **-f** (forever : örökké) opciója, amely kiegészíti a **tail** funkcióit a **readslow**-éival. Magyarázzuk meg ezt a tervezést! □

7-2. feladat. Mi történik a **readslow**-val, ha a fájlt olvasás közben megcsonkítják? Hogyan lehetne ezt kezelni? Útmutatás: olvassuk el a 7.3. szakaszban az **fstat**-ról szóló részt! □

Fájllétrehozás: open, creat, close, unlink

Az alapértelmezés szerinti szabványos bemeneti, kimeneti és hibafájlok kivételével meg kell nyitnunk azokat a fájlokat, amelyeket olvasni vagy írni akarjuk. Ehhez két rendszerhívás, az **open** és a **creat** áll rendelkezésünkre.*

* Ken Thompsont megkérdezték, hogy ha újratervezné a UNIX rendszert, mit csinálna másképp. Ezt válaszolta: „A **creat** végére kiírnám az **e**-t."

Az **open** olyan, mint az előző fejezet **fopen**-je, csak most a fájlmutató helyett fájlleírót kapunk vissza, ami **int** típusú szám.

```
char *név;
int fd, rwmode;
fd = open(név, rwmode);
```

A név argumentum itt is, akárcsak az **fopen** esetében, a fájl nevét tartalmazó karaktersorozat. Az elérési mód argumentum viszont más, mint az **fopen**-nél volt. Az **rwmode** értéke olvasás esetén 0, írás esetén 1, és ha egy fájlt írásra *és* olvasásra is meg akarunk nyitni, akkor 2. Az **open** −1-et ad vissza, ha a műveletben valamilyen hiba volt, egyébként egy érvényes fájlleírót.

Hiba az is, ha egy nem létező fájlt akarunk megnyitni. Új fájlt a **creat** rendszerhívás segítségével lehet létrehozni vagy egy régit kell felülírni.

```
int engedélyek;
fd = creat(név, engedélyek);
```

Ha a fájl a kívánt néven létrehozható, akkor a **creat** a fájlleírót adja vissza, ha nem, akkor −1-et. Ha létrehozható, akkor a **creat** létrehozza az engedélyek argumentumban megadott hozzáférési jogosultsággal. Ha a fájl már létezett − ez nem hiba −, akkor a tartalmát a **creat** törli, azaz hosszát 0-ra csökkenti. (Az engedélyeket azonban nem változtatja meg.) Az **engedélyek** értékétől függetlenül a **creat**-tel éppen létrehozott fájl írásra meg van nyitva.

Mint a 2. fejezetben leírtuk, kilenc bit szolgál a védelmi információ tárolására, az olvasási, az írási és végrehajtási jog ellenőrzésére. Ez a kilenc bit egy háromjegyű oktális számmal írható le. Pl. a 0755 a tulajdonos részére írást, olvasást és végrehajtást, a csoportbeliek és többiek részére olvasást és végrehajtást engedélyez. Ne felejtsük el a vezető 0-t, amely a C nyelvben az oktális számokat jelzi!

Példaként tekintsük a **cp** leegyszerűsített változatát! A leglényegesebb egyszerűsítés, hogy a mi változatunk csak egy fájlt másol és második argumentuma nem lehet jegyzék. Egy másik gyenge pontja változatunknak az, hogy nem másolja a forrásfájl engedélyeit. Később megmutatjuk, hogyan lehet ezen segíteni.

```
/* cp: legegyszerűbb változat */
#include <stdio.h>
#define PERMS 0644 /* RW a tulajdonosnak, R a csoportnak és a többieknek */
char *progname;
main (argc, argv) /* cp: f1-et az f2-be másolja */
  int argc;
  char *argv[];
{
  int f1, f2, n;
  char buf[BUFSIZ];
  progname = argv[0];
  if (argc != 3)
    error("Usage: %s from to", progname);
```

```
  if ((f1 = open(argv[1], 0)) == -1)
    error("can't open %s", argv[1]);
  if ((f2 = creat(argv[2], PERMS)) == -1)
    error("can't create %s", argv[2]);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
    if (write(f2, buf, n) != n)
      error ("write error", (char *) 0);
  exit(0);
}
```

Az **error** rutint a következőkben fogjuk tárgyalni.

A program által egy időben megnyitható fájlok száma korlátozott (általában 20 körül van, a pontos értékhez keressük meg a **<sys/param.h>**-ban a **NOFILE**-t!). Ennek megfelelően a sok fájlt használó programokat úgy kell megírni, hogy képesek legyenek újra használni a fájlleírókat. A **close** rendszerhívás megszünteti a fájlnév és a fájlleíró egymáshoz rendelését, így felszabadítja a fájlleírót egy másik fájl számára. Ha egy program **exit** utasítással befejeződik, vagy visszatér a hívó programba, akkor az összes megnyitott fájl lezáródik.

Az **unlink** rendszerhívással lehet a fájlt törölni a fájlrendszerből.

Hibakezelés: errno

Az ebben a fejezetben tárgyalt rendszerhívások – valójában az összes rendszerhívás – futhat hibára. A −1 értékkel való visszatérés csak a hiba tényét jelzi. Néha azonban a hiba okát is szeretnénk tudni. A rendszerhívások ehhez egy hibaszámot tesznek az **errno** külső egész típusú változóba. (A hibaszámok jelentése megtalálható a UNIX kézikönyv 2. fejezetének bevezetésében.) Az **errno** felhasználásával a program eldöntheti, hogy a megnyitás hibájának oka pl.: nem létezik a fájl, vagy nincs olvasásra engedélyünk. Rendelkezésünkre áll még a **sys_errlist** nevű karakterlánctömb is, amelyből az **errno**-val indexelve kiválasztható a hibaszámhoz tartozó magyarázószöveg. Az **error** rutin általunk írt változata is ezeket az adatszerkezetet használja:

```
error(s1, s2) /* kiirja a hibaüzenetet és meghal */
  char *s1, *s2;
{
  extern int errno, sys_nerr;
  extern char *sys_errlist[], *progname;
  if (progname)
    fprintf(stderr, "%s: ", progname);
fprintf(stderr, s1, s2);
if (errno > 0 && errno < sys_nerr)
    fprintf(stderr, " (%s)", sys_errlist[errno]);
fprintf(stderr, "\n");
  exit(1);
}
```

Az **errno** kezdőértéke 0, és mindig kisebb kell, hogy legyen, mint a **sys_nerr**. A hibátlan rendszerhívás az **errno** értékét nem állítja nullára, ezért minden hiba előfordulása után nullázni kell, ha azt akarjuk, hogy programunk továbbra is jól működjön.

Nézzük meg, hogy a **cp** jelenlegi változatában a hibák hogy jelennek meg:

```
$ cp foo bar
cp: can't open foo (No such file or directory)
$ date >foo; chmod 0 foo          Letiltjuk az olvasást.
$ cp foo bar
cp: can't open foo (Permission denied)
$
```

Közvetlen elérés: lseek

A fájlbemenet és -kimenet általában szekvenciális, minden **read**, ill. **write** a fájlnak pontosan arra a helyére vonatkozik, ahol az előző parancs abbahagyta. A fájlt azonban lehet tetszés szerinti sorrendben is olvasni vagy írni. Az **lseek** rendszerhívással a fájlban az olvasás vagy írás tényleges végrehajtása nélkül is mozoghatunk:

```
int fd, kezdet;
long offset, pos, lseek();
pos = lseek(fd, offset, kezdet);
```

Példánkkal az **fd** fájlleíróhoz tartozó fájlban a **kezdet** argumentumban megadott helyhez képest **offset**-ban adott pozícióval előbb vagy hátrább levő aktuális pozícióra állunk. A soron következő írás vagy olvasás az aktuális pozícióban kezdődik. A **kezdet** értéke 0, 1 vagy 2 lehet attól függően, hogy az **offset**-et a fájl elejétől, az aktuális helyzettől vagy a fájl végétől mérjük. A visszaadott érték az új abszolút pozíció, ill. hiba esetén -1. Pl. fájlhoz hozzáíráskor az írást megelőző fájl végére állás:

```
lseek(fed, 0L, 2);
```

A fájl elejére állás (rewind):

```
lseek(fd, 0L, 0);
```

Az aktuális helyzet meghatározása:

```
pos = lseek(fd, 0L, 1);
```

Figyeljük meg a **0L** argumentumot: az **offset, long** (hosszú) típusú egész. (Az **lseek** névben az **l** betű a *long* (hosszú) rövidítése; így különböztetjük meg a UNIX 6. kiadásában szereplő **seek** rendszerhívástól, amely argumentumaiban rövid egészekkel dolgozik.)

Az **lseek** segítségével a fájlok többé-kevésbé úgy kezelhetők, mint a nagy tömbök,

de ennek ára a lassabb hozzáférés. Pl. a következő függvény akárhány bájtot be tud olvasni a fájl bármelyik helyéről.

```
get(fd, pos, buf, n) /* pos pozíciótól kezdve n bájtot olvas */
    int fd, n;
    long pos;
    char *buf;
{
    if (lseek(fd, pos, 0) == -1) /* pozícionálás */
        return -1;
    else
        return read(fd, buf, n);
}
```

7-3. feladat. Módosítsuk a **readslow** programot úgy, hogy kezelje a fájlnévargumentumot is, ha megadjuk! Továbbá vezessünk be egy **-e** opciót:

```
$ readslow -e
```

Ennek hatására a **readslow** keresse meg a bemeneti fájl végét, mielőtt olvasni kezdene. Mit csinál az **lseek** egy csővel? □

7-4. feladat. Írjuk át a 6. fejezet **efopen**-jét úgy, hogy hívja meg az **error**-t! □

7.2. Fájlrendszer; jegyzékek

Következő témánk az, hogy hogyan sétálhatunk végig a jegyzékhierarchián. Ehhez nem kell újabb rendszerhívás, csak néhány eddig megismert, de új szerepkörben. Példaként készítsünk egy **spname** nevű függvényt, amely a félregépelt fájlneveket próbálja javítani. Az

```
n = spname(név, újnév);
```

függvény keres egy, a **név** argumentumban adott nevet „jól közelítő" fájlnevet. Ha talált ilyet, akkor ezt az **újnév** argumentumban adja vissza. Az **spname** visszatérési értéke: -1, ha nem talált eléggé hasonlót; 0, ha pontosan rátalált a fájlra; 1, ha egy eltérőt választott ki.

Az **spname** jól alkalmazható a **p** parancs kibővítésére: ha ki akarunk iratni egy fájlt és elgépeltük a nevét, a **p** megjelenítheti a **név** jó közelítését, és megkérdezhetné, nem erre gondoltunk-e:

```
$ p /urs/srx/ccmd/p/spnam.c        Borzalmasan összetákolt név.
"/usr/src/cmd/p/spname.c"? y       Elfogadtuk a javasolt változtatást.
/* spname: a helyesen írt névvel tér vissza */
...
```

Az **spname**-t úgy írjuk meg, hogy a fájlnév minden egyes összetevőjében próbálja meg kijavítani a következő típusú hibákat: egy karakter elhagyása, ill. hozzáadása; egy karakter tévesztése; két karakter felcserélése. Mindez szerepelt az előbbi példában. Az ilyen program valóságos áldás a szórakozottan gépelők számára.

Mielőtt leírnánk a programot, helyénvaló egy rövid áttekintés a fájlrendszer felépítéséről. A jegyzék egy olyan fájl, amelyben a fájlok neve és a fájlok elhelyezkedésére utaló információ található. Az elhelyezkedésre utaló információ egy táblázatra vonatkozó index. Ezt a táblázatot nevezzük *i-bög-táblának*. Egy fájl i-bögében a fájlnév kivételével mindent nyilvántartunk a fájlról. Így a jegyzék egy sora csak két dolgot tartalmaz, egy i-bög-számot és a fájl nevét. A pontos leírás a **<sys/dir.h>** nevű fájlban található meg:

```
$ cat /usr/include/sys/dir.h
#define DIRSIZ 14 /* a fájlnév max. hossza */
struct direct     /* a jegyzék egy sorának szerkezete */
{
  ino_t d_ino; /* i-bög száma */
  char d_name[DIRSIZ]; /* fájlnév */
};
$
```

Az **ino_t** típus az i-bög-táblázat indexének **typedef**-je. A PDP−11-en és a VAX-on levő rendszerváltozatokban ez történetesen **unsigned short**, de ez gépenként eltérő lehet, így ezt az információt semmiképp sem szabad beépítenünk programunkba. Ezért van a **typedef**! Az ehhez hasonló *rendszerfüggő* típusok teljes készlete a **<sys/types.h>**-ban található, amelyet programunkba a **<sys/dir.h>** használata előtt be kell építenünk.

Az **spname** működése elég áttekinthető, annak ellenére, hogy igen sok peremfeltételre kell tekintettel lenni. Tegyük fel, hogy a fájl neve **/d1/d2/f**. A keresés alapgondolata az, hogy a keresett fájlnévből először lehámozzuk az első összetevőt (/), majd keressünk a jegyzékben egy olyan bejegyzést, amelynek a neve a következő összetevőhöz (d1) hasonló, ezután azt a bejegyzést keressük meg, amelyik a d2 jó közelítése, és így tovább az összes összetevőt egyeztetjük. Amennyiben a jegyzékben valamelyik összetevőhöz nem találunk megfelelő jelöltet, a keresést nem folytatjuk.

A feladatot három függvénnyel oldjuk meg. Maga az **spname** elkülöníti az összetevőket és felépíti az „addig legjobban illeszkedő" fájlnevet, azaz az aktuális sejtést. Közben hívja a **mindist** függvényt, amely megkeresi az adott jegyzékben az aktuális sejtéshez legjobban illeszkedő fájlnevet, felhasználva a harmadik, **spdist** nevű függvényt, amely meghatározza a két név közti eltérést.

```
/* spname: a helyesen írt névvel tér vissza */
/*
 * spname(réginév, újnév) char *réginév, *újnév;
 * visszatérés −1 ha a régi névhez nincs hasonló,
 *              0 ha teljes az egyezés,
 *              1 ha közelítő nevet talált.
 * a javított nevet az újnévben tárolja.
 */
```

```
#include <sys/types.h>
#include <sys/dir.h>
spname(réginév, újnév)
  char *réginév, *újnév;
{
  char *p, guess[DIRSIZ+1], best[DIRSIZ+1];
  char *new = újnév, *old = réginév;
  for (;;) {
    while (*old == '/') /* törtvonalon átlép*/
      *new++ = *old++;
    *new = '\0';
    if (*old == '\0') /* azonos vagy javított */
      return strcmp(réginév, újnév) != 0;
    p = guess; /* a következő összetevőt a sejtésbe másolja */
    for ( ; *old != '/' && *old != '\0'; old++)
      if (p < guess+DIRSIZ)
        *p++ = *old;
    *p = '\0';
    if (mindist(újnév, guess, best) >= 3)
      return -1; /* reménytelen */
    for (p = best; *new = *p++; ) /* kiegészítés */
      new++;                      /* az újnévhez */
  }
}

mindist(dir, guess, best) /* jegyzékkeresés a sejtéshez */
  char *dir, *guess, *best;
{
  /* best beállítása, távolság 0...3 */
  int d, nd, fd;
  struct {
      ino_t ino;
      char name[DIRSIZ+1]; /* dir.h-beli + 1 */
  } nbuf;
  nbuf.name[DIRSIZ] = '\0'; /* '\0' miatt a visszatérés:+1 */
  if (dir[0] == '\0' /* aktuális jegyzék */
      dir = ".";
  d = 3; /* legkisebb eltérés */
  if ((fd=open(dir, 0)) == -1)
    return d;
  while (read(fd,(char *) &nbuf,sizeof(struct direct)) > 0)
    if (nbuf.ino) {
      nd = spdist(nbuf.name, guess);
      if (nd <= d && nd != 3) {
        strcpy(best, nbuf.name);
        d = nd;
        if (d == 0) /* pontos egyezés */
        break;
      }
    }
```

```
    close(fd);
    return d;
}
```

Ha a **mindist**-nek átadott jegyéknév üres, akkor pont (.) nevűt keresünk. A **mindist** egyszerre csak egy jegyzékelemet olvas be. Vegyük észre, hogy a **read** puffere egy struktúra, nem pedig egy karaktertömb. A **sizeof** függvénnyel számoljuk a bájtok számát, amely meghatározza a karaktermutató címértékét.

A jegyzék használaton kívüli sorát (pl. kitörölt fájl helyét), amelynek az i-bög-bejegyzése nulla, átlépjük. Az eltérés ellenőrzésére nem az

```
if (nd < d ...)
```

feltételt használjuk, hanem az

```
if (nd <= d ...)
```

feltételt, tehát minden jegyzék jobb egyezés, mint a ., amely mindig a jegyzék első bejegyzése.

```
/* spdist: két név közti eltérést adja meg */
/*
 * az eltérések durva mértéke
 * 0, ha azonos a két szöveg
 * 1, ha két karakter felcserélődött
 * 2, ha egy karakter hibás, hiányzik vagy töröltük
 * 3, egyébként
 */

#define EQ(s,t) (strcmp(s,t) == 0)
spdist(s, t)
    char *s, *t;
{
    while(*s++ == *t)
        if (*t++ == '\0')
            return 0; /* pontos egyezés */
    if (*--s) {
        if (*t) {
            if (s[1] && t[1[ && *s == t[1]
                && *t == s[1] && EQ(s+2, t+2))
                    return 1; /* helycsere */
            if (EQ(s+1, t+1))
                return 2; /* 1 karaktereltérés */
        }
        if (EQ(s+1, t))
            return 2; /* extra karakter */
    }
```

```
    if (*t && EQ(s, t+1))
      return 2; /* hiányzó karakter */
    return 3;
  }
```

Ha már van egy **spname** függvényünk, nem nehéz azt a **p** programba beépíteni:

```
/* p: bemenetét darabonként írja ki (4. változat) */

#include <stdio.h>
#define PAGESIZE     22
char *progname; /* a program neve a hibaüzenetekhez */
main(argc, argv)
  int argc;
  char *argv[];
{
  FILE *fp, *efopen();
  int i, pagesize = PAGESIZE;
  char *p, *getenv(), buf[BUFSIZ];
  progname = argv[0];
  if ((p=getenv("PAGESIZE")) != NULL)
    pagesize = atoi(p),
  if (argc > 1 && argv[1][0] == '-') {
    pagesize = atoi(&argv[1][1]);
    argc--;
    argv++;
  }
  if (argc == 1)
    print(stdin, pagesize);
  else
    for (i = 1; i < argc; i++)
      switch (spname(argv[i], buf)) {
      case -1: /* nem egyezhet */
        fp = efopen(argv[i], "r");
        break;
    case 1: /* javítva */
      fprintf(stderr, "\"%s\"? ", buf);
      if (ttyin() == 'n')
        break;
      argv[i] = buf; /* füstbement...*/
    case 0: /* teljes egyezés */
      fp = efopen(argv[i], "r");
      print(fp, pagesize);
      fclose(fp);
    }
  exit(0);
}
```

A fájlnevek javítását nem minden fájlneveket feldolgozó programba célszerű beépíteni. Az interaktív **p** programban jól működik, de nem használható olyan programokban, amelyek nem csak párbeszédes üzemmódban dolgoznak.

7-5. feladat. Mennyire lehet javítani az **spname** programban a legjobban egyező név kiválasztására használt módszert? Pl. ostobaság egy közönséges fájlt úgy kezelni, mintha jegyzék volna, ez pedig előfordulhat a jelenlegi változatban. □

7-6. feladat. Ha a jegyzék végén a **tx** nevet találjuk, akkor ez minden **t**-vel kezdődő kétkarakteres nevet jól közelít. Tudnánk-e az eltérésnek mérőszámot találni? Valósítsuk meg és próbáljuk ki, hogy mennyire működik jól a gyakorlatban! □

7-7. feladat. A **mindist** egyszerre csak egy bejegyzést olvas be a jegyzékből. Érzékelhetően gyorsabb lesz-e a **p** program, ha a jegyzékből egyszerre többet olvasunk? □

7-8. feladat. Változtassuk meg az **spname**-t! Ha nem talál egy névhez közel állót, akkor olyan nevet keressen, amelyiknek a folytatása megegyezik a kiinduló név elejével. Mi a teendő, ha így több név is megfelelő lenne? □

7-9. feladat. Milyen programokban használhatjuk még az **spname** gondolatait? Tervezzünk egy önálló programot, amely javítja saját argumentumait, mielőtt átadja azokat egy másik programnak! Például így:

```
$ fix prog fájlnevek...
```

Meg tudnánk írni a **cd** egy olyan változatát, amely használja az **spname** programot? Hogyan lehetne a rendszerbe beépíteni? □

7.3. Fájlrendszer; i-bögök

Ebben a szakaszban a fájlrendszer információira vonatkozó rendszerhívásokat tárgyaljuk. Különösen azokat, amelyek fájlok jellemzőivel: méret, időpont, engedély stb. kapcsolatosak. Ezek a rendszerhívások teszik elérhetővé a 2. fejezetben ismertetett információkat.

Először kukkantsunk bele az i-bögbe. Az i-bög egy részét a **<sys/stat.h>**-ban definiált **stat** struktúra írja le:

```
struct stat /* a stat-ban található struktúra */
{
  dev_t     st_dev;      /* i-bög-eszköz */
  ino_t     st_ino;      /* i-bög-szám */
  short     st_mode;     /* módbitek */
  short     st_nlink;    /* a láncszemek száma */
  short     st_uid;      /* a tulajdonos azonosítója */
```

```
    short       st_gid;         /* a csoport azonosítója */
    dev_t       st_rdev;        /* különleges fájlok részére */
    off_t       st_size;        /* fájlméret karakterekben */
    time_t      st_atime,       /* az utolsó olvasás időpontja */
    time_t      st_mtime,       /* az utolsó írás vagy értelmezés időpontja */
    time_t      st_ctime,       /* a fájl vagy i-bög utolsó változásának időpontja */
};
```

A mezők többségének értelmezése a megjegyzésrészben megtalálható. A típusok, mint pl. **dev_t** és **ino_t** definíciói, a korábban elmondottaknak megfelelően a **<sys/types.h>** fájlban vannak. Az **st_mode** mező a fájlt leíró jelzőket tartalmazza, amelyek jelentése az egyszerűség kedvéért szintén megtalálható a **<sys/stat.h>** fájlban:

```
#define S_IFMT      0170000   /* a fájl típusa */
#define S_IFDIR     0040000   /* jegyzék */
#define S_IFCHR     0020000   /* különleges karakter */
#define S_IFBLK     0060000   /* különleges blokk */
#define S_IFREG     0100000   /* szabályos */
#define S_ISUID     0004000   /* felhasználói azonosító a futtatáshoz */
#define S_ISGID     0002000   /* csoportazonosító a futtatáshoz */
#define S_ISVTX     0001000   /* módosított szöveg megőrzése */
#define S_READ      0000400   /* a tulajdonos olvasási engedélye */
#define S_IWRITE    0000200   /* a tulajdonos írási engedélye */
#define S_EXEC      0000100   /* a tulajdonos végrehajtási engedélye */
```

Egy fájl i-bögét a **stat** és az **fstat** rendszerhívásokkal lehet lekérdezni. A **stat** a fájlnév alapján visszaadja az i-bög-adatokat, vagy −1-et, ha valami hiba volt. Az **fstat** ugyanezt teszi egy nyitva levő fájl fájlleírója (nem pedig egy **FILE** mutatója) alapján. Tehát

```
char *név;
int fd;
struct stat stbuf;
stat(név, &stbuf);
fstat(fd, &stbuf);
```

a fájlnév, ill. az **fd** fájlleíró alapján az **stbuf** struktúrába beírja az i-bög információit.

Ennyi ismeret birtokában már elkezdhetünk használható programot írni. Kezdjük a postaládát figyelő **checkmail** program C nyelvű változatával. Ha a postafájl nagyobb lesz, akkor a **checkmail** programunk kiírja a „You have mail" üzenetet és megszólaltatja a csengőt. (A fájl rövidülhet is, de ilyenkor nem kell üzenetet küldeni, mivel feltehetően csak néhány elolvasott levelet töröltünk.) Kezdetnek ez a megközelítés tökéletesen megfelel, ha majd már működik, lehet szépíteni rajta.

```
/* checkmail: postaláda-figyelés */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
char *progname;
char *maildir = "/usr/spool/mail"; /* rendszerfüggő */
main(argc, argv)
  int argc;
  char *argv[];
{
  struct stat buf;
  char *name, *getlogin();
  int lastsize = 0;
  progname = argv[0];
  if ((name = getlogin()) == NULL)
    error("can't get login name", (char *) 0);
  if (chdir(maildir) == -1)
    error("can't cd to %s", maildir);
  for (; ;) {
    if (stat(name, &buf) == -1) /* nincs levélszekrény */
      buf.st_size = 0;
    if (buf.st_size > lastsize)
      fprintf(stderr, "\nYou have mail\007\n");
    lastsize = buf.st_size;
    sleep(60),
  }
}
```

A **getlogin(3)** függvény a bejelentkezési nevünket adja vissza, vagy NULL-t, ha nevünket nem találja. A **checkmail** a **chdir** rendszerhívással a levél jegyzékére áll, így a következő **stat** hívásnak már nem kell a gyökértől kiindulva az összes jegyzéket végigkeresnie. Olvasóinknak a **maildir**-t esetenként meg kell változtatnia ahhoz, hogy az megfeleljen a rendelkezésükre álló UNIX rendszernek. A **checkmail**-t úgy írtuk, hogy a keresést akkor is megismételje, ha korábban nem volt levélszekrényünk, ugyanis a **mail** legtöbb változata a kiürült levélszekrényt azonnal törli.

Ezt a programot az 5. fejezetben, a burok ciklusainak bemutatása során már megírtuk. Az első változat mindig több folyamatot indított, amikor belenézett a levélszekrénybe és ezzel a kelleténél több terhet rótt a rendszerre. A C nyelvű változat egyetlen folyamatból áll, amely percenként csak egyszer hívja a **stat**-ot. Vajon mibe kerül, ha a **checkmail**-t állandóan futtatjuk a háttérben? Méréseink szerint egy óra alatt egy másodpercnél jóval kevesebbet igényel, ami igazán nem számottevő.

A hibakezelés bemutatása: sv

Következő példánk egy **sv** nevű, a **cp**-hez hasonló másolóprogram. Ez is egy fájlsorozatot másol át egy jegyzékbe, de a célfájlt csak akkor írja felül, ha az a forrásnál régebbi, ill. ha még nem létezett a forrásnak megfelelő célfájl a jegyzékben, akkor létrehozza a célfájlt. Az **sv** elgondolásunk szerint mentőprogram, azaz nem írja felül a régebbi változattal azt, amelyik újabbnak tűnik.

Az **sv** több információt használ fel az i-bögből, mint a **checkmail**.

Azt tervezzük, hogy a

$ *sv fájl1 fájl2 ... dir*

átmásolja a **fájl1**-et a **dir/fájl1**-be, a **fájl2**-t **dir/fájl2**-be stb. Kivéve, ha a **dir/fájl** újabb, mint a **fájl**, akkor másolás helyett az **sv** figyelmeztető üzenetet ad. El akarjuk kerülni a láncolt fájlok többszörös átmásolását, ezért a forrásfájlnevekben nem engedjük meg a törtvonal (/) karaktert.

```
/* sv: új fájlok átmentése */
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>
char *progname;
main(argc, argv)
  int argc;
  char *argv[];
{
  int i;
  struct stat stbuf;
  char *dir = argv[argc-1];
  progname = argv[0];
  if (argc <= 2)
    error("Usage:%s files... dir", progname);
  if (stat(dir, &stbuf) == -1)
    error("can't access directory %s", dir);
  if ((stbuf.st_mode & S_IFMT) != S_IFDIR)
    error("%s is not a directory", dir);
  for (i = 1; i < argc-1; i++)
    sv(argv[i], dir);
  exit(0);
}
```

Az i-bögökben az időt az 1970. január 1. 0 : 00 GMT óta eltelt másodpercek számával mérjük, tehát a régebbi fájlok **st_mtime** mezőjében kisebb értékek vannak.

```
sv(file, dir) /* fájl átmentése a dir jegyzékbe */
  char *file, *dir,
{
  struct stat sti, sto;
  int fin, fout, n;
  char target[BUFSIZ], buf[BUFSIZ], *index();
  sprintf(target, "%s/%s", dir, file);
  if (index(file, '/') != NULL) /* némely rendszerben strchr() */
    error("won't handle /'s in %s", file);
  if (stat(file, &sti) == -1)
    error("can't stat %s", file);
  if (stat(target, &sto) == -1) /* nincs célfájl */
    sto.st_mtime = 0; /* réginek látszik, létrehozzuk */
```

```
    if (sti.st_mtime < sto.st_mtime) /* a cél az újabb */
        fprintf(stderr, "%s: %s not copied\n",
            progname, file);
    else if ((fin = open(file, 0)) == -1)
        error("can't open file %s", file);
    else if ((fout = creat(target, sti.st_mode)) == -1)
        error("can't create %s", target);
    else
        while ((n = read(fin, buf, sizeof buf)) > 0)
            if (write(fout, buf, n) != n)
                error("error writing %s", target);
    close(fin);
    close(fout);
}
```

A szabványos be- és kimeneti függvények helyett azért használtuk a **creat**-et, hogy az **sv** átörökíthesse a bemenetifájl-módot is. (Megjegyezzük, hogy az **index** és az **strechr** ugyanannak a rutinnak a névváltozatai. A UNIX kézikönyvben a string(3) címszó alatt lehet megnézni, hogy az adott rendszer melyik nevet használja.)

Bár az **sv** program eléggé egyedi, néhány fontos gondolatot sugall. Sok program nem rendszerprogram, de használ olyan információt, amelyet az operációs rendszer kezel, és amely csak rendszerhívásokon keresztül érhető el. Az ilyen programok készítésénél igen lényeges tudni, hogy az információ csak az olyan szabványos fejléc fájlokban jelenik meg, mint amilyenek a **<stat.h>** és **<dir.h>**, és hogy a programok ezeket a fájlokat építik be, nem pedig magukat az aktuális deklarációkat. Az ezt figyelembe vevő program sokkal nagyobb valószínűséggel lesz átvihető az egyik rendszerből a másikba.

Azt is érdemes észrevenni, hogy az **sv** program legalább a kétharmada hibakezelés. A programírás első szakaszában nagy a kísértés, hogy a hibakezelést elhanyagoljuk, mivel ez eltérít a fő feladattól. S ha egyszer egy program már „működik", akkor kevés erőnk marad arra, hogy beszúrjuk azokat az ellenőrzéseket, amelyek a programot bármilyen körülmények között is működőképessé teszik.

Az **sv** nincs minden baj ellen felvértezve (pl. nem figyel a rosszkor jövő megszakításokra), de a legtöbb programnál elővigyázatosabb. Csak egyetlen részletet emeljünk ki egy pillanatra; az utolsó **write** utasítást! Ritkán fordul elő, hogy egy **write** nem sikerül, ezért nagyon sok program nem készül fel erre a lehetőségre. A lemezen azonban elfogyhat a hely, a felhasználó túllépheti az engedélyezett területet, az összeköttetés megszakadhat, és ez mind íráshibát okozhat. Sokkal jobban járunk, ha a program kijelzi ezeket a hibákat és nem tesz úgy, mintha minden rendben menne.

A tanulság az, hogy a hibakezelés fárasztó, de fontos dolog. Az ebben a könyvben szereplő legtöbb programmal kapcsolatban egyrészt a helyszűke miatt, másrészt az érdekesebb dolgokra való összpontosítás miatt nagyvonalúak voltunk. Az igazi, a termék szintű programokban nem szabad a hibákat elhanyagolni!

7-10. feladat. Változtassuk meg a **checkmail** programot úgy, hogy azonosítsa az üzenet küldőjét és ezzel egészítse ki a „You have mail" üzenetet! Útmutatás: **sscanf**, **lseek**. □

7-11. feladat. (Nehéz.) Változtassuk meg a **checkmail** programot úgy, hogy a ciklusba lépés előtt ne kapcsoljon át a postajegyzékhez! Lényegesen megváltoztatja ez a futási időt? Meg tudjuk-e írni a **checkmail** egy olyan változatát, ami egyetlen folyamattal az összes felhasználót értesíti? □

7-12. feladat. Írjunk egy **watchfile** programot, amely egy fájlt figyel, és annak minden változásakor kiírja a tartalmát elejétől végéig! Mikor tudnánk ezt hasznosítani? □

7-13. feladat. Az **sv** hibakezelése meglehetősen rugalmatlan. Módosítsuk úgy, hogy tovább dolgozzon akkor is, ha nem tudott feldolgozni egy fájlt! □

7-14. feladat. Tegyük az **sv**-t rekurzívvá: ha a forrásfájlok egyike egy katalógus, akkor ezt a katalógust és a hozzá tartozó fájlokat hasonló módon dolgozza fel! Tegyük a **cp**-t is rekurzívvá! Elemezzük, hogy a **cp** −**v** és az **sv** ugyanúgy működik-e, hiszen a **cp** −**v** sem másol át olyan fájlt, amelynél a célfájl frisebb! □

7-15. feladat. Írjunk egy

```
$ random fájlnév
```

programot, amely találomra kiválaszt egy sort a fájlból! Ha adott a neveket tartalmazó **emberek** fájl, akkor a **random** jól használható egy bűnbakkereső programban, amellyel a felelős gyakorlatilag pontosan meghatározható:

```
$ cat bűnbak
echo "Mindennek `random emberek` az oka!"
$ bűnbak
Mindennek Kis János az oka!
$
```

Ellenőrizzük, hogy a **random** a sorok hosszának tetszőleges eloszlása esetén hogyan működik! □

7-16. feladat. Az i-bög még további adatokat is tárol, pl. lemezcímeket, ahol a fájlblokkok találhatók. Nézzük át a **(sys/ino.h)** fájlt, és írjunk egy **icat** programot, amely elolvassa az i-bög-szám és a lemezegység által megadott fájlt! (Természetesen csak akkor fog működni, ha a szóban forgó lemez elolvasható. Milyen körülmények között hasznosíthatnánk az **icat** programot? □

7.4. Folyamatok

Ebben a szakaszban bemutatjuk, hogyan lehet egy programból egy másikat futtatni. Ennek legegyszerűbb módja a **system** szabványos könyvtári rutin, amelyet a 6. fejezetben már említettünk, de nem részleteztünk. A **system**-nek egy parancssorból álló argumentuma van, amelyik megegyezik a terminálon beírt parancssorral (csupán

a végére írt újsor-karakter hiányzik). A **system** az argumentumát egy alburokban végrehajtja. Ha a parancssort részekből kell összerakni, akkor hasznos lehet a **sprintf** tárbeli formázási lehetősége. E szakasz végén bemutatjuk a **system** interaktív programokhoz készült, megbízhatóbb változatát, de először lássuk azokat az összetevőket, amelyekből majd felépül ez a program!

Alacsony szintű folyamatok létrehozása: execlp és execvp

A legalapvetőbb tevékenység egy programból egy másik program *visszatérés nélküli* végrehajtása, amit az **execlp** rendszerhívással érhetünk el. Pl. ahhoz, hogy egy futó program utolsó tevékenységeként kiírja a dátumot, az

```
execlp("date", "date", (char *) 0);
```

hívást lehet használni. Az **execlp** első argumentuma a parancs fájlneve, az **execlp** a mi környezetünkből (a **$PATH**-ból) veszi a keresési utat, és ugyanúgy keres, mint a burok. A második és az azt követő argumentumok a parancsnév és a névvel adott parancs argumentumai, ezek kerülnek az új program **argv** tömbjébe. A lista végét a **0** argumentum jelzi. (Az exec(2) elolvasásával bepillantást kapunk az **execlp** tervezésébe.)

Az **execlp** hívás a futó programra egy újat hív, lefuttatja azt, majd kilép. Az eredeti program csak valamilyen hiba esetén kapja vissza a vezérlést, pl. ha a megadott fájl nem található vagy nem végrehajtható:

```
execlp("date", "date", (char *) 0);
fprintf(stderr, "Couldn't execute 'date'\n");
exit(1);
```

Az **execlp** egyik változata az **execpv**, amelyet akkor használunk, ha nem tudjuk előre az argumentumok számát. Hívása:

```
execvp(fájlnév, argp);
```

ahol **argp** az argumentumok mutatóinak tömbje (mint az **argv**), a tömb utolsó mutatója kötelezően a **NULL** mutató, az **execvp** ebből ismeri fel a lista végét. Akárcsak az **execlp** esetében, a **fájlnév** annak a fájlnak a neve, amelyben a program található, az **argp** az új program **argv** tömbje, **argp[0]** pedig a program neve.

Ezen rutinok közül egyik sem terjeszti ki az argumentumlistát az olyan metakarakterekre, mint a <, >, *, idézőjelek stb. Ha ezeket akarjuk használni, akkor az **execlp**-vel a **/bin/sh** burkot kell meghívnunk, amely elvégzi a kért tevékenységeket. Állítsunk össze egy parancssorszöveget, amely a teljes parancsot tartalmazza olyan formában, mintha a terminálon gépeltük volna be, majd adjuk ki az

```
execlp("/bin/sh", "sh", "-c", parancssor, (char *) 0);
```

parancsot! A **-c** argumentum azt jelenti, hogy a következő argumentumot egy teljes parancssorként, nem pedig különálló argumentumként kell kezelni.

Az **exec** bemutatására tekintsük a **waitfile** programot. A

$ *waitfile fájlnév [parancs]*

parancssor rendszeres időközönként ellenőrzi a megnevezett fájlt. Ha az utolsó ellenőrzés óta nem változott, akkor végrehajtja az argumentumként adott parancsot. Ha a fájlnév után nem áll **parancs**, akkor a fájlt a **waitfile** a szabványos kimenetre másolja. A **waitfile** programot a **troff** menetének figyelésére használhatjuk:

$ *waitfile troff.out echo troff done &*

A **waitfile** program megvalósítása az **fstat** felhasználásával állapítja meg, hogy mikor változott utoljára a fájl:

```
/* waitfile: vár, amíg a fájl már nem változik */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
char *progname;
main(argc, argv)
  int argc;
  char *argv[];
{
  int fd;
  struct stat stbuf;
  time_t old_time = 0;
  progname = argv[0];
  if (argc < 2)
    error("Usage: %s filename [cmd]", progname);
  if ((fd = open(argv[1], 0)) == -1)
    error("can't open %s", argv[1]);
  fstat(fd, &stbuf);
  while (stbuf.st_mtime != old_time) {
    old_time = stbuf.st_mtime;
    sleep(60);
    fstat(fd, &stbuf);
  }
  if (argc == 2) { /* fájlmásolás */
    execlp("cat", "cat", argv[1], (char*) 0);
    error("can't execute cat %s", argv[1]);
  } else {      /* futtatási folyamat */
    execvp(argv[2], &argv[2]);
    error("can't execute %s", argv[2]);
  }
  exit(0);
}
```

Ezzel mind az **execlp**, mind az **execvp** használatára mutattunk példát.

Azért választottuk példaként ezt a tervezést, mert jól használható, de más változatok is lehetségesek, pl. a **waitfile** egyszerűen visszatérhet, amikor fájl már nem módosul.

7-17. feladat. Módosítsuk a **watchfile**-t (l. a 7-12. feladatot) úgy, hogy a **waitfile**-hoz hasonlóan működjön! Ha nincs *parancs*, akkor másolja át a fájlt; egyébként hajtsa végre a parancsot! A **watchfile** és a **waitfile** tartalmazhatnak-e azonos forráskódot? Útmutatás: **argv[0]**. □

Folyamatok ellenőrzése: fork és wait

A következő lépés: az **execlp** vagy **execvp**-vel futtatott program futtatása után a vezérlés visszaszerzése. Mivel ezek a rutinok egyszerűen a régi program helyére behozzák az újat, a régi megőrzéséhez először is két példány kellene belőle. Ezek közül az első helyére jönne az új, míg a másik várna a bejött program futásának befejeződésére. A megduplázást a **fork** rendszerhívás végzi el:

```
proc_id = fork();
```

hatására két példány lesz a programból, s mindegyik tovább fut. A két példány közti egyetlen eltérés a **fork** által visszaadott folyamat azonosító értéke. Az egyik folyamatban (a *gyermekben*) a **proc_id** nulla. A másikban (a *szülőben*) a **proc_id** nem nulla; ez a gyermekfolyamat azonosítója. Így egy program visszatéréssel való meghívásának alapvető eljárása a következő:

```
if (fork() == 0)
  execlp("/bin/sh", "sh", "-c", parancssor, (char *) 0);
```

És valóban, a hibakezelés kivételével ez elegendő is. A **fork** két példányt készít a programból. A gyermekben a **fork** által visszaadott érték nulla, tehát ez meghívja az **execlp**-et, amely végrehajtja a parancssort és meghal. A szülőprogramban a **fork** nemnulla értékkel tér vissza, így ez átugorja az **execlp**-et. (Hiba esetén a **fork** −1-et ad vissza.)

Általában a szülő megvárja a gyermek futásának befejeződését, mielőtt folytatná saját működését. Ezt a **wait** rendszerhívással éri el:

```
int status;

if (fork() == 0)
  execlp(...);          /* gyermek */
wait(&status);          /* szülő */
```

Ez még nem kezel semmilyen rendellenes állapotot, mint pl. az **execlp** vagy **fork** hibás visszatérését, vagy annak lehetőségét, hogy egynél több gyermek fut párhuzamosan. (A **wait** a befejeződött gyermekfolyamat azonosítóját visszaadja, ezáltal összevethet-

jük azt és a **fork** által visszaadott értéket.) Végül ez a töredék nem foglalkozik a gyermek esetleges rendhagyó viselkedésével sem, mégis, ez a három sor a szabványos **system**-hívás lényege.

A **wait** által visszaadott **status** alsó nyolc bitjén kódolódik a rendszer elképzelése a gyermek kilépési státusáról; normális befejeződés esetében ez nulla, a nemnulla érték valamilyen rendellenességet jelez. A következő magasabb helyi értékű nyolc bit az **exit** hívásból vagy a **main**-ből a gyermekfolyamat befejeződése által okozott visszatérés argumentumából származik.

Ha a burok hív meg egy programot, akkor a **0**, **1** és **2** fájlleírók a megfelelő fájlra mutatnak, és az összes többi fájlleíró felhasználható. Amikor ez a program meghív egy másikat, akkor az illendőség azt követeli, hogy ugyanezek a feltételek teljesüljenek. Sem a **fork**-, sem az **exec**-hívás nincs semmilyen hatással a nyitott fájlokra, a szülő és a gyermek ugyanazon nyitott fájlokkal rendelkezik. Ha a szülő olyan kimenetet pufferez, amelynek a gyermek kimenete előtt ki kell kerülnie, akkor a szülőnek az **execlp**-hívás előtt ki kell íratnia a puffert. Megfordítva, ha a szülő a bemenő áradatot pufferezi, akkor a gyermek számára mindaz az információ elvész amit a szülő már beolvasott. A kimenetet ki lehet küldeni, de a bemenetet nem lehet visszaállítani. Mindkét szempontot figyelembe kell venni, ha a be- és kimenetet a 6. fejezetben tárgyalt szabványos függvényekkel oldjuk meg, mivel ezek általában mind a bemenetet, mind a kimenetet pufferezik.

Az **execlp**-en keresztül öröklődő fájlleírók megtörik a **system** programot: ha a hívó program szabványos be- és kimenete nincs a képernyőhöz hozzárendelve, akkor a **system** által hívott parancsé sem lesz. Lehet, hogy pont erre van szükségünk; az **ed** szövegszerkesztő esetében pl. egy felkiáltójellel (!) kezdődő parancs bemenetének valószínűleg a szövegből kell jönnie. Még **ed**-nek is karakterenként kell olvasnia a bemenetét a pufferezési hibák elkerülése végett.

A **p**-hez hasonlóan interaktív programok számára viszont a **system**-nek vissza kell kapcsolnia a szabványos be- és kimenetet a terminálra. Az összekapcsolás egyik módja a **/dev/tty**.

A **dup(fd)** rendszerhívás az **fd** fájlleírót a legkisebb még ki nem osztott fájlleíróval megduplázza, az újonnan visszaadott fájlleíró ugyanarra a nyitott fájlra vonatkozik. A következő programrészlet a program szabványos bemenetét egy fájlhoz kapcsolja:

```
int fd;

fd = open("fájl", 0);
close(0);
dup(fd);
close(fd);
```

A **close(0)** felszabadítja a **0**-ás fájlleírót, azaz a szabványos bemenetet, de mint már megszoktuk, ez nem befolyásolja a szülőt.

A következőkben bemutatjuk a **system** azon változatát, amely interaktív programokhoz is megfelelő; hibaüzenetekhez a **progname**-t használja. A jelekkel foglalkozó részeket kihagyhatjuk, erre a következő szakaszban fogunk visszatérni.

```
/*
 * A system biztonságosabb változata interaktív programokhoz
 */
#include <signal.h>
#include <stdio.h>
system(s)      /* az s parancssor futtatása */
  char *s;
{
  int status, pid, w, tty;
  int (*istat)(), (*qstat)();
  extern char *progname;
  fflush(stdout);
  tty = open("/dev/tty", 2);
  if (tty == -1) {
    fprintf(stderr, "%s: can't open /dev/tty\n", progname);
    return -1;
  }
  if ((pid = fork()) == 0) {
    close(0); dup(tty);
    close(1); dup(tty);
    close(2); dup(tty);
    close(tty);
    execlp("sh", "sh", "-c", s, (char *) 0);
    exit(127);
  }
  close(tty);
  istat = signal(SIGINT, SIG_IGN);
  qstat = signal(SIGQUIT, SIG_IGN);
  while ((w = wait(&status)) != pid && w != -1)
    ;
  if (w == -1)
    status = -1;
  signal(SIGINT, istat);
  signal(SIGQUIT, qstat);
  return status;
}
```

Megjegyezzük, hogy a **/dev/tty**-t 2 móddal – írásra és olvasásra – nyitottuk meg, és így duplikáltuk a szabványos be- és kimenethez. Valójában bejelentkezéskor a rendszer is így gyűjti össze a baüzenetet. Ezután már írhatunk a szabványos bemenetre:

```
$ echo hello 1>&0
hello
$
```

Ez azt jelenti, hogy a **2**-es fájlleíró duplikálásával összeköthetnénk a szabványos be- és kimenetet, de a **/dev/tty** megnyitás tisztább és biztonságosabb. Még ez a **system** program is rejt hibákat magában: a hívó nyitott fájljait, mint amilyen pl. a **p** program **ttyin** rutinjának **tty** fájlja, át fogja adni a gyermekfolyamatnak.

Nem arról kívántuk meggyőzni Olvasóinkat, hogy a mi **system**-változatunkat kell használniuk összes programukban – minden bizonnyal lefullasztana pl. egy nem párbeszédes **ed**-et –, hanem azt akartuk bemutatni, hogy hogyan tarthatjuk kézben a folyamatokat, és hogyan használhatjuk *helyesen* az egyszerű elemeket. A helyesen alkalmazásonként mást és mást jelenthet, és nem biztos, hogy megegyezik a **system** szokásos megvalósításával.

7.5. Jelek és megszakítások

A szakaszban azzal foglalkozunk, hogy hogyan lehet elegánsan kezelni a külvilágból származó jeleket (mint pl. a megszakításokat) és a programhibákat. A programhibák főleg meg nem engedett tárterületre való hivatkozásokból, különös utasítások végrehajtásából és lebegőpontos hibákból erednek. A leggyakoribb külső jelek: a *megszakítás,* ami a *DEL* gomb megnyomásakor keletkezik; a *kilépés,* amit az *FS* karakter (*ctl*–\) *állít elő; a vonalszakadás,* amit a telefon lerakása okoz, és a *befejeződés,* amit a **kill** parancs állít elő. Amikor az előző események közül valamelyik bekövetkezik, akkor az arról a terminálról indított összes folyamat kap egy jelet, és hacsak másképpen nem rendelkeztünk, a jel befejezteti a folyamatot. A legtöbb jel hatására a program tárterülete a nyomkövetés céljára egy lemezfájlba íródik. (L. **adb(1)** és **sdb(1)**!)

A **signal** rendszerhívás megváltoztatja az alapértelmezés szerinti kezelést. Két argumentuma van. Az első egy szám, amely azonosítja a jelet. A második vagy egy függvény címe, vagy egy kód, amely a jel figyelmen kívül hagyását vagy az alapértelmezés szerinti tevékenység végrehajtását kéri. A különböző argumentumok leírását a **<signal.h>** fájl tartalmazza. Így az

```
#include <signal.h>
...
signal(SIGINT, SIGINT_IGN);
```

kikapcsolja a jelek figyelését, míg a

```
signal(SIGINT, SIG_DFL);
```

visszaállítja az alapértelmezés szerinti tevékenységet, a folyamat befejezését. Minden esetben visszatérésekor a **signal** a jel előző értékét adja vissza. Ha a **signal** második argumentuma egy függvény neve (amelyet ugyanebben forrásfájlban korábban már deklarálni kellett), akkor a jel előfordulásakor meg fog hívódni ez a függvény. Legtöbbször ezt a lehetőséget használjuk fel, hogy befejeződése előtt a program be nem fejezett tevékenységében rendet teremthessen, pl. kitörölhessen egy ideiglenes fájlt:

```
#include <signal.h>
char *tempfile = "temp.XXXXXX";
```

```
main()
{
  extern onintr();
  if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, onintr);
  mktemp(tempfile);

  /* Folyamat... */

  exit(0);
}
onintr() /* megszakításkor rendet csinál */
{
  unlink(tempfile);
  exit(1);
}
```

Miért van a **main**-ben ellenőrzés és kétszeres **signal**-hívás? Emlékezzünk vissza, hogy a jeleket a terminálról indított összes folyamat megkapja! Ennek megfelelően, ha egy programot nem interaktívan (& jellel indítva) futtatunk, akkor a burok elrendezi, hogy a program figyelmen kívül hagyja a jeleket, így nem fog leállni az előtérben futó folyamatoknak szánt jelek miatt. Ha a program azt bejelentve indul, hogy az összes jelet feltétel nélkül az **onintr** rutinnak kell megkapnia, akkor ez megszünteti a buroknak azt az erőfeszítését, hogy a háttérben való futtatáskor megvédje azt.

A bemutatott megoldás az, hogy a jelkezelés állapotát ellenőrizzük, és továbbra is figyelmen kívül hagyjuk a jeleket, ha korábban azokat már figyelmen kívül hagytuk. Az így megírt kód függ attól, hogy a **signal**-hívás visszaadja-e ennek a jelnek az előző állapotát. Ha a jeleket már figyelmen kívül hagytuk, a folyamatnak továbbra is figyelmen kívül kell hagynia, egyébként pedig el kell fognia azokat.

Egy még kifinomultabb program esetleg arra készül fel, hogy elfogjon egy jelet és azt úgy értelmezze, hogy abba kell hagynia, amit éppen csinál, majd vissza kell térnie a saját parancsfeldolgozó ciklusához. Gondoljunk egy szövegszerkesztőre: egy hosszú kiíratás megszakítása nem okozhat kilépést, elveszítve ezzel az addig végzett munkát. Erre az esetre lehet a következő programot írni:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;
main()
{
  int onintr();
  if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, onintr);
  setjmp(sjbuf); /* kimenti a veremmutatót */
  for (; ;) {
    /* fő feldolgozási ciklus */
  }
  ...
```

```
}
onintr() /* megszakításkor töröl */
{
   signal(SIGINT, onintr); /* töröl a következő megszakítás számára */
   printf("\nInterrupt\n");
   longjmp(sjbuf, 0); /* visszatér a kimentett állapothoz */
}
```

A **<setjmp.h>** fájlban a **jmp_buf** típusdeklaráció olyan, amelyben a verempozíció megőrizhető; az **sjbuf**-of ilyennek deklaráltuk. A **setjmp(3)** függvény kiment egy rekordot a program pillanatnyi állapotáról. A változók értékeit nem menti ki. Megszakítás esetén meghívódik az **onintr** rutin, amely üzenetet írhat ki, jelzőket állíthat be stb. Amit a **setjmp** tárolt, argumentumként kezeli a **longjmp**, majd a **setjmp** hívás utáni helyre adja vissza a vezérlést. Így a vezérlés (és a veremszint) visszaáll a főrutinban arra a helyre, ahol a főciklus elkezdődött.

Megjegyezzük, hogy **onintr**-ben a megszakítás előfordulásakor a jelet újra beállítjuk. Ezt meg kell tenni, mivel a jelek előfordulásukkor automatikusan visszakapják az alapértelmezés szerinti értéküket.

Bizonyos programok, amelyek a jeleket érzékelni akarják, egyszerűen nem állíthatók meg tetszőleges ponton, pl. egy bonyolult adatszerkezet megváltoztatása közben. A megoldás az, hogy a megszakítási rutin csak beállít egy jelzőt, és visszatér ahelyett, hogy **exit** vagy **longjmp** hívást adna ki. A végrehajtás pontosan ott folytatódik, ahol a megszakítás történt, és a megszakítási jelző később megvizsgálható.

Ezzel a megközelítéssel kapcsolatban azonban van egy kis baj. Tegyük fel, hogy a program éppen a terminálról olvas, amikor megszakítás érkezik. A megadott rutin illendően meghívódik, beállítja a jelzőt és visszatér. Ha valóban igaz, amit korábban mondtunk, akkor a végrehajtás „pontosan ott folytatódik, ahol a megszakítás történt", a program folytatná a terminálról olvasást, amíg a felhasználó begépel egy újabb sort. Ez a viselkedés eléggé megtévesztő lenne, mivel a felhasználó nem tudhatja, hogy a program éppen olvas, és feltehetően jobban örülne, ha a jel azonnal kifejtené a hatását. Ennek érdekében a rendszer befejezi a **read**-et, de egy hibastátust ad vissza, ami jelzi, hogy mi történt: **errno** az **<errno.h>** fájlban definiált **EINTR**-re áll be, jelezve, hogy megszakított rendszerhívás fordult elő.

Azokat a programokat, amelyek elfogják a jeleket, majd folytatják a végrehajtást, fel kell készíteni a megszakított rendszerhívások által okozott „hibák" kezelésére. (A rendszerhívások figyelik a terminálról érkező **read**-eket, **wait**- és **pause**-hívásokkal várakoznak.) Egy ilyen program a következőhöz hasonló kódot használhat a szabványos bemenetről való olvasáskor:

```
#include <errno.h>
extern int errno;
...
if (read(0, &c, 1) <= 0)      /* EOF vagy megszakítás */
   if (errno == EINTR) { /* megszakítás által okozott EOF */
      errno = 0;  /* helyreállítás a következő alkalomra */
      ...
```

```
    } else {      /* valódi EOF */
        ...
}
```

Van még egy figyelemre méltó finomság, ha a jelek elfogását más programok végrehajtásával kombináljuk. Tegyük fel, hogy a program elfogja a megszakítást és ugyanakkor ismer egy módszert (mint az **ed**-ben a !), amellyel másik programokat lehet végrehajtani. Ilyen esetben a kód így nézhetne ki:

```
if (fork() == 0)
    execlp(...);
signal(SIGINT, SIG_IGN);        /* a szülő elengedi a megszakítást */
wait(&status);                  /* amíg a gyermek dolgozik */
signal(SIGINT, onintr);         /* a megszakítást újra kezeli */
```

Miért van ez így? A jeleket megkapja az összes folyamatunk. Tegyük fel, hogy az általunk meghívott program elfogja a saját megszakításait, mint ahogy ezt egy szövegszerkesztő teszi. Ha megszakítjuk az alprogramot, akkor az érzékelni fogja a jelet és visszatér a főciklusához, feltehetően azért, hogy olvasson a terminálról. De a hívó program is ki fog lépni az alprogramra való **wait**-jéből és olvasni fog a terminálról. Nagyon zavaró, ha két folyamat olvas a terminálunkról, mert olyan a hatása, mintha pénzfeldobással döntené el a rendszer, hogy az egyes bemenő sorokat melyik folyamat kapja meg. A megoldás az, hogy a szülőprogram hagyja figyelmen kívül a megszakításokat mindaddig, amíg a gyermek befejeződik. Ez az okoskodás tükröződik vissza a **system** jelkezelésében is:

```
#include <signal.h>
system(s)   /* az s parancssort futtatja */
    char *s;
{
    int status, pid, w, tty;
    int (*istat)(), (*qstat)();
    ...
    if ((pid = fork()) == 0) {
        ...
        execlp("sh", "sh", "-c", s, (char *) 0);
        exit(127);
    }
    ...
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return status;
}
```

A deklarációkból egyébként kiderül, hogy a **signal** függvénynek elég különös második argumentuma van. Valójában ez a mutató egy egész számot adó függvényre mutat, és ez a jel rutin típusa is. A **SIG_IGN** és **SIG_DFL** értékeknek megvan a megfelelő típusa, amelyek választásuk szerint nem eshetnek egybe egyetlen lehetséges aktuális függvénnyel sem. A lelkesebbek kedvéért bemutatjuk ezek a PDP−11-en és a VAX-on levő definícióit: ezek a definíciók valószínűleg elég elrettentőek ahhoz, hogy a **<signal.h>** használatát tartsuk előnyösebbnek.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

Riasztás

Az **alarm(n)** rendszerhívás egy SIGALRM jelet küld folyamatunknak n másodperc elteltével. A riasztásjel segítségével megbizonyosodhatunk arról, hogy valami bekövetkezett-e az adott időn belül; ha az esemény bekövetkezett, akkor a riasztásjel kikapcsolható, de ha nem, akkor a folyamat a riasztásjel elfogásával visszaszerezheti a vezérlést.

Példaként tekintsünk egy **timeout** nevű programot, amely egy másik parancsot futtat! Ha ez a parancs nem fejeződik be az adott időn belül, akkor az a riasztás pillanatában törlődik. Emlékezzünk vissza a **watchfor** parancsra az 5. fejezetből! Ahelyett, hogy korlátozás nélkül futtatnánk, beállíthatunk egy egyórás időhatárt:

```
$ timeout −3600 watchfor dmg &
```

A **timeout** programja szinte mindenre támaszkodik, amiről az utolsó két szakaszban beszéltünk. Létrehozunk egy gyermeket, a szülő beállít egy riasztást, majd vár a gyermek befejeződésére. Ha a riasztás érkezik először, akkor törli a gyermeket, és megkísérli a gyermek kilépési státusát visszaadni.

```
/* timeout: a folyamathoz időkorlátot szab */
#include <stdio.h>
#include <signal.h>
int pid; /* a gyermekfolyamat azonosítója */
char *progname;
main(argc, argv)
  int argc;
  char *argv[];
{
  int sec = 10, status, onalarm();
  progname = argv[0];
  if (argc > 1 && argv[1][0] == '−') {
    sec = atoi(&argv[1][1]);
    argc−−;
    argv++;
```

```
    }
    if (argc < 2)
        error("Usage: %s [-10] command", progname);
    if ((pid=fork()) == 0) {
        execvp(argv[1], &argv[1]);
        error("couldn't start %s", argv[1]);
    }
        signal(SIGALRM, onalarm);
    alarm(sec);
    if (wait(&status) == -1 || (status & 0177) != 0)
        error("%s killed", argv[1]);
    exit((status >> 8)& 0377);
}
onalarm()       /* riasztáskor törli a gyermeket */
{
    kill(pid, SIGKILL);
}
```

7-18. feladat. Hogyan valósítható meg a **sleep** függvény? Útmutatás: **pause(2)**. Milyen körülmények között fordulhat elő, ha egyáltalán előfordulhat, hogy a **sleep** és az **alarm** összegabalyodnak? □

Történeti és irodalmi megjegyzések

A UNIX rendszer megvalósításának nincs részletes leírása, részben azért, mert a program tulajdonosi védelem alatt áll. Ken Thompsonnak a „UNIX implementation" című tanulmánya (BSTJ, 1978. július) leírja az alapgondolatokat. Más, hasonló témájú cikkek: „The UNIX system – a retrospective" a *BSTJ* azonos számában és „The evolution of the UNIX time-sharing system" (Symposium on Language Design and Programming Methology, Springer-Verlag, *Lecture Notes in Computer Science* # 79, 1979.). Mindkettőt Dennis Ritchie írta.

A **readslow** programot Peter Weinberger arra találta ki, hogy Ken Thompson és Joe Condon sakkozógépének, a Belle-nek működését a játszmák alkalmával kis többletterheléssel figyelhessék a nézők. Belle a játék állását egy fájlba írta, a kibicek a **readslow** segítségével úgy tudták kilesni a fájlt, hogy alig raboltak el valamit Belle drága idejéből. (A Belle-hardver legújabb változata a központi gépen alig végez számítást, tehát a probléma megszűnt.)

Az **spname** ötletét Tom Duff sugallta nekünk. Ivor Durham, David Lamb és James Saxe „Spelling correction in user interface" című tanulmányában (CACM 1983. október) egy, a miénktől némileg eltérő megközelítést mutat az elírási hibák javítására egy postaprogram kapcsán.

8. Programfejlesztés

A UNIX rendszert eredetileg programfejlesztési környezetnek szánták. Ebben a fejezetben néhány, a programfejlesztés során különösen jól használható eszközről lesz szó. Egy, a BASIC-kel összemérhető programnyelv értelmezőprogramjának kidolgozását kísérjük majd figyelemmel. Azért választottuk ezt, mert megvalósítása során találkozhatunk mindazokkal a nehézségekkel, amelyek nagy programok létrehozásakor előfordulnak. Továbbá sok programot érdemes úgy felfogni, mint egy programnyelv értelmezőjét, amely jól összeállított bemenetre tényked̈ések és kimenetek sorozatával válaszol. Nézzük tehát a programnyelvek fordításához, értelmezéséhez használható programfejlesztési eszközöket!

Ebben a fejezetben a következőkre térünk ki:

- **yacc** egy szintaktikai elemzőt előállító, azaz egy olyan program, amely a nyelv nyelvtani leírása alapján előállítja annak elemzőjét;
- **make** egy olyan program, amely meghatározza és vezérli egy bonyolult program fordításának folyamatát;
- **lex** egy olyan, a **yacc**-hoz hasonló program, amely lexikai elemzőket készít.

Az ilyen jellegű feladatok kezelésére vonatkozóan van még néhány fontos észrevételünk: kis feladatot megoldó programmal kezdjünk, majd ezt hagyjuk kiterebélyesedni; figyeljünk a nyelv fejlődésére; használjuk ki az eszközeink lehetőségeit.

A programnyelvet hat lépcsőben fogjuk megvalósítani. Mindegyik lépcső eredménye már használható lenne akkor is, ha a fejlesztés nem folytatódna. A programot annak idején mi is ehhez hasonló fokozatokban írtuk meg. A lépcsők a következők:

1. Olyan négy alapműveletet ismerő „számológépet" készítünk, amely: lehetővé teszi a +, −, *, / és a zárójelek használatát; lebegőpontos számokkal dolgozik; egy sorba egy kifejezést lehet begépelni; a kifejezések értékét azonnal kiírja.

2. Ez a változat elfogad egykarakteres változóneveket, de ezek még csak betűk lehetnek. Már ismeri a negatív előjeleket, és rendelkezik némi hibaelhárítással is.

3. Az ebben a lépcsőben elkészülő program megenged tetszőleges hosszúságú változóneveket, és ismeri a beépített függvényeket (sin, exp stb.), valamint olyan hasznos állandókat, mint a π (a karakterkészlet korlátai miatt **PI**-ként írva), és még egy műveletet, a hatványozást.

4. Ebben a fázisban programunk működési elvét változtatjuk: minden egyes utasításhoz kódot generálunk, majd ezt értelmezzük, szemben a korábbi, menet

közbeni kiértékeléssel. A felhasználó számára tehát most nincs továbbfejlesztés, de ez a lépés vezet el az 5. lépcsőhöz.

5. Kiegészítjük programnyelvünket programvezérlési eszközökkel: **if-else** és **while**, { és } jelekkel összekapcsolt utasításokkal, relációs műveletekkel, mint pl. >, <, = stb.

6. Lehetővé tesszük az argumentumokkal rendelkező rekurzív függvények és eljárások, továbbá szövegeket és számokat beolvasó és kiíró utasítások használatát.

A végleges nyelvet a 9. fejezetben adjuk meg. Ez lesz egyben könyvünkben a UNIX dokumentum készítésének legátfogóbb példája is. (A referenciakönyv a 2. függelékben található meg.)

A fejezet igen hosszú, mivel egy komolyabb program becsületes megírása során nagyon sok részletet kell érintenünk ahhoz, hogy érthető legyen, amit mondunk. Feltételezzük, hogy Olvasónk érti a C nyelvet, és az esetleg homályos részleteket – pl. a UNIX programozói kézikönyv második kötete segítségével – tisztázni tudja, hiszen egyszerűen nincs annyi papírunk, hogy minden részletre kitérjünk. Gyűrkőzzünk neki ennek a fejezetnek, és ne lepődjünk meg, ha többször is el kell majd olvasnunk! Megadtuk a program végleges változatának teljes listáját is a 3. függelékben, így jobban áttekinthető, hogyan illeszkednek egymáshoz a különböző részletek.

Ami pedig a program elnevezését illeti, rengeteg időt elpocsékoltunk a névválasztásra, de igazán jót mégsem találtunk. Végül is a **hoc**-nál maradtunk, ami a „high order calculator"-ra, azaz a magasabb rendű számológépre utal. Az egyes változatok nevei ebből adódóan: **hoc1**, **hoc2** stb.

8.1. Első lépcső: a négyműveletes számológép

Ebben a részben a **hoc1** egyik változatát mutatjuk be. Ez a program a legegyszerűbb zsebszámológépek szolgáltatásaival körülbelül azonos lehetőségeket nyújt, de azoknál lényegesen nehezebben hordozható. Csak négy műveletet ismer, úgymint: +, −, *, /, de megengedi a tetszőleges mélységig egymásba ágyazott zárójelek használatát, amivel kevés zsebszámológép dicsekedhet. Ha egy begépelt kifejezést *RETURN* követ, akkor az eredmény a következő sorban azonnal megjelenik:

```
$ hoc1
4*3*2
        24
(1+2) * (3+4)
        21
1/2
        0.5
355/113
        3.1415929
-3-4

hoc1: syntax error near line 4      A negatív számot még nem ismeri!
$
```

Nyelvtanok

Amióta az ALGOL nyelv leírására kifejlesztették a Backus – Naur-féle formát, azóta a nyelveket formális nyelvtanok írják le. A **hoc1** nyelvtana elvont formájában kicsi és egyszerű:

```
list:   expr \n
        list expr \n
expr:   NUMBER
        expr + expr
        expr - expr
        expr * expr
        expr / expr
        ( expr )
```

Más szóval a lista (**list**) kifejezéseknek (**expr**) a sorozata, és mindegyik listát egy újsor-karakter zár le. Egy kifejezés lehet: egy szám (**NUMBER**); műveleti jellel egybekapcsolt két kifejezés; egy zárójelbe tett kifejezés.

A leírás így még nem teljes. Többek között nem írtuk le a műveletek alapértelmezés szerinti végrehajtási sorrendjét, és nem értelmeztük egyik szerkezetet sem. A listát meghatároztuk ugyan a kifejezéssel, a kifejezést pedig a **NUMBER** segítségével, magát a **NUMBER**-t sehol sem határoztuk meg. Ezekre a részletekre is ki kell térni ahhoz, hogy a nyelv vázlatos leírásától eljuthassunk egy működő programhoz.

A yacc áttekintése

A **yacc** egy elemzőprogramot előállító program*, amely egy nyelvnek az előbbiekhez hasonló leírása alapján egy olyan programot készít, amelyik ezen a nyelven írt utasításokat képes elemezni. A **yacc** teremt lehetőséget arra, hogy a nyelvtani szerkezetekhez jelentést is rendeljünk, méghozzá oly módon, hogy az elemzéssel együtt az utasítások jelentése is „kiértékelhető" legyen. A **yacc** használatában a következő fokozatokat különböztetjük meg.

Először egy nyelvtant írunk, hasonlóképpen az előbbiekhez, csak pontosabban. Ez megadja a nyelv szintaktikáját. Ezen a fokon a nyelvtani leírásban meglevő hibák és kétértelmű részek kiszűrésére használható a **yacc**.

Másodszor a nyelvtan minden egyes szabályához vagy termékéhez hozzárendelünk egy *tevékenységet*, amit a programunknak el kell végeznie, ha az elemzés alatt álló programban ezt a nyelvtani szerkezetet találja. A „mi a teendő" részt C nyelven írjuk, szokásunk szerint a C nyelvű listához kapcsolva a nyelvtant. Így adjuk meg a nyelv szemantikáját.

Harmadszor készítünk egy lexikai elemzőt, amely beolvassa az elemzés alatt álló programot, és a további elemzés számára értelmes egységekre bontja. Pl. egy ilyen értelmes egység a **NUMBER**, amely több karakter hosszú; vagy az egykarakternyi

* Az **yacc** "yet another compiler compiler"-t jelent, amivel szerzője, Steve Johnson arra utalt, hogy megírásának idején (1972 körül) számos ilyen jellegű program létezett. Az **yacc** egy a sok hasonló közül.

műveleti jelek (pl. a + és −). A lexikai egységeket hagyományosan szimbólumoknak, angolul **token**-nek nevezik.

Végezetül egy vezérlőrutint készítünk, amely a **yacc** által felépített elemzőt hívja meg. A **yacc** a nyelvtan és a „tevékenységek" alapján előállít egy **yyparse**-nek nevezett elemzőt, és kiírja egy C nyelvű fájlba. Ha a **yacc** nem talál hibát, akkor az elemzőt, a lexikai elemzőt és a vezérlőrutint lefordíthatjuk, más C nyelvű rutinokkal egybeszerkeszthetjük és végrehajthatjuk. Ez a program úgy működik, hogy a tokenekre ismételten meghívja a lexikai elemzőt, felismeri a nyelvtani (szintaktikus) szerkezetét és végrehajtja a felismert nyelvtani szabályhoz tartozó szemantikus tevékenységeket. A lexikai elemző belépési pontját nevezzük **yylex**-nek, mivel ezt a funkciót az **yyparse** minden alkalommal meghívja, ha egy újabb tokent szeretne kapni. (Az összes olyan név, amit a **yacc** használ, **y**-nal kezdődik.)

Hogy kicsit pontosabbak legyünk, a **yacc** bemenete a következő:

```
%{
C utasítások, mint pl. #include, deklarációk stb. Ez a rész opcionális.
%}
yacc deklarációk: tokenek, nyelvtani változók, a precedencia és az asszociativitás szabályai
%%
nyelvtani szabályok és tevékenységek
%%
egyéb C utasítások (opcionális):
main() { ... ; yyparse(); ... }
yylex() { ... }
...
```

Ezeket a **yacc** dolgozza fel és az eredményt az **y.tab.c** nevű fájlba írja, amelynek a felépítése a következő:

```
%{ és %} közötti C nyelvű utasítások (ha voltak),
a második %% utáni C nyelvű utasítások (ha voltak):
main() { ...; yyparse(); ... }
yylex() { ... }
...
yyparse() { egy elemző, amelyik az yylex()-et hívja }
```

A UNIX-ra jellemzően a **yacc** program C nyelvű fájlt állít elő, nem pedig tárgykódút (**.o**). Ez a legrugalmasabb megközelítés – az így előállított kód gépfüggetlen és alkalmas arra, hogy egy jó ötlet esetén bármikor újra felhasználjuk egy másik feldolgozásban.

A **yacc** önmagában is egy hasznos eszköz. Meg kell tanulni, ami nem könnyű, de az erőfeszítés sokszorosan megtérül. A **yacc** programmal készített elemző kicsi, hatékony és jól működik (bár szemantikai tevékenységeiért mi vagyunk a felelősek); rengeteg utólagos elemzési problémát kezel automatikusan. Így könnyűszerrel hozhatók létre a programnyelvek felismerőprogramjai, és (ami ennél is fontosabb) ezek a nyelv fejlesztése során könnyűszerrel újra és újra módosíthatók.

A **hoc1** forrásprogramja a következő részekből áll: a nyelvtani leírásból, amely a teendőket is tartalmazza; az **yylex** lexikai rutinból; valamint a **main** főprogramból. Mindez a **hoc.y** fájlban található. (A **yacc** által kezelt fájlok nevei hagyományosan **.y**-ra végződnek, de ezt a **yacc** nem követeli meg, nem úgy, mint a **cc** program a **.c**-t.) A nyelvtani rész teszi ki **hoc.y** első felét:

```
$ cat hoc.y
%{
#define YYSTYPE double /* a yacc verem adattípusa. */
%}
%token  NUMBER
%left   '+' '-' /* balról asszociatív, alacsony precedencia */
%left   '*' '/' /* balról asszociatív, magasabb precedencia */
%%
list:   /* üres */
        | list '\n'
        | list expr '\n'        { printf("t%.8g\n", $2); }
        ;
expr:   NUMBER                  { $$ = $1; }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         { $$ = $1 / $3; }
        | '('expr ')'           { $$ = $2; }
        ;
%%      /* nyelvtan vége */
...
```

Ebbe a pár sorba nagyon sok új információt szorítottunk be. Nem fogunk mindent elmagyarázni, és természetesen azt sem, hogy maga az elemző hogyan működik – ez megtalálható a **yacc** kézikönyvben.

A függőleges vonal (| jel) a vagylagosan alkalmazható szabályokat választja el. Bármelyik nyelvtani szabályhoz hozzá lehet rendelni egy műveletet, amelyiket a **yacc** akkor hajt végre, ha a nyelvtani szabályt alkalmazhatónak találta. Egy művelet a { és } (kapcsos zárójel) közé zárt C utasítások sorozatát jelenti. Egy művelet leírásában a **$n** (azaz **$1**, **$2** stb.) a szabály **n**-edik összetevőjének az értékét jelenti. **$$** pedig a teljes szabály kiértékelésekor kapott érték. Tehát pl. az

```
expr: NUMBER { $$ = $1; }
```

szabályban a **$1** a felismert **NUMBER** értékével azonos; az így kapott érték a kifejezés értéke. A **$$ = $1** értékadás elhagyható – a **$$** mindig **$1** értékét kapja, ha mást nem adtunk meg.

A következő szabály,

```
expr: expr '+' expr { $$ = $1 + $3; }
```

az összetett kifejezés értéke a két összetevő kifejezés összegével azonos. Megjegyzendő, hogy '+' jelenti a **$2**-t, ugyanis minden összetevő sorszámot kap.

Az e feletti szint egy kifejezés, amelyet új sor ('\n') követ. Ezt listaként kell értelmezni, és ki kell írni az értékét. Ha a bemenő sor vége ilyen szerkezetű, akkor az elemzés eredményesen fejeződik be. A lista üres is lehet; így kezeljük az üres bemenő sorokat.

A **yacc** bemenete szabad formátumú; a mi formátumunk csak egy ajánlott szabvány.

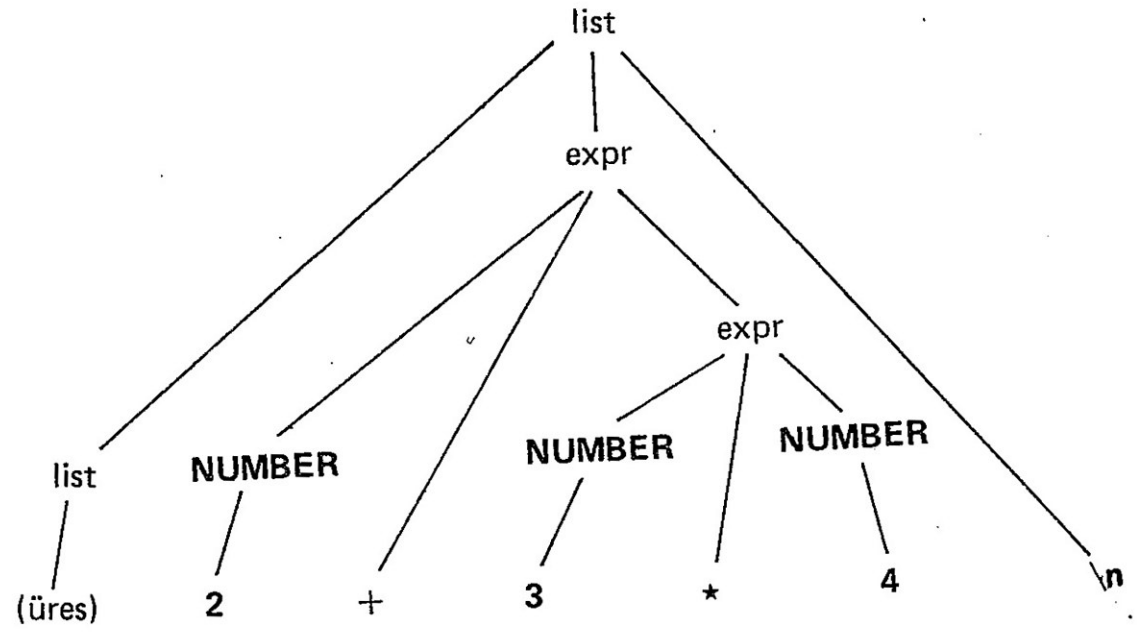
Ebben a változatban a bemenet felismerése, ill. kielemzése a kifejezés azonnali kiértékelésére vezet. Bonyolultabb helyzetekben (ide értve **hoc4**-et és leszármazottait) az elemzési folyamat később végrehajtandó kódot állít elő.

Szemléletesebbé tehetjük az elemzést a 8.1. ábrán bemutatotthoz hasonló elemzési fa felrajzolásával. Úgy képzelhetjük el, hogy az elemző az értékeket a levelektől a gyökér felé haladva számítja ki és adja át.

A szabályokkal előállított értékek menet közben a verembe kerülnek; ennek alapján lehet értéket átadni az egyik szabályból a következőnek. A verem adattípusa általában egész (**int**), de mivel a lebegőpontos adatokkal is dolgozunk, meg kell változtatnunk az alapértelmezést. A verem típusát a

```
#define YYSTYPE double
```

definíció duplapontosnak állítja be.



8.1. ábra. A 2+3*4 elemzőfája

A lexikai elemző által felismerendő szintaktikus egységeket deklarálni kell, kivéve az olyan egy jellel leírtakat, mint a '+' és '−'. A **%token** deklarációval egy vagy több ilyen egységet lehet deklarálni. A bal vagy a jobb oldali asszociativitást is kiköthetjük, ehhez a **%token** helyett a **%left**-et, ill. a **%right**-ot kell használnunk. (A bal-asszociativitás azt jelenti, hogy az **a−b−c** kifejezés kiértékelési sorrendje **(a−b)−c** és nem **a−(b−c)**.) A precedenciaszabályt a megadás sorrendje határozza meg: ugyanabban a deklarációban szereplő műveleti jelek azonos precedenciaszinten vannak; a később megadott műveleti jelnek magasabb szintű a precedenciája. Ily módon a tulajdonképpeni nyelvtan nem egyértelmű (azaz bizonyos bemenő karaktersorozatot többféle módon lehet értelmezni), de a deklarációban megadott kiegészítő információk feloldják a kétértelműséget.

A **hoc.y** fájl második részében levő rutinok alkotják a program hátralevő részét:

hoc.y folytatása

```
#include <studio.h>
#include <ctype.h>
char    *progname;          /* hibaüzenetek részére */
int     lineno = 1;
main(argc, argv)            /* hoc1 */
    char *argv[];
{
    progname = argv[0];
    yyparse();
}
```

A **main** az **yyparse** programmal elemezteti ki a bemenetet. A kifejezések egymás utáni feldolgozása teljes egészében a nyelvtani elemzés része, amit a listákat felépítő szabályok segítenek. (Ehelyett azt is tehettük volna, hogy a **main** programbeli **yyparse** hívás körül egy kis ciklust szervezünk, és ebbe vesszük bele a lista értékének kiíratását.)

Az **yyparse** viszont az **yylex**-et hívja a bemenő tokenek mindegyikére. A mi **yylex**-ünk egyszerű: a szóköz- és tabulátorkaraktereket átlépi, a számjegyláncokhoz számértéket rendel, a hibaüzenetekhez sorszámozza a bemenő sorokat, és minden más karaktert változatlanul ad vissza. Mivel a nyelvtan csak a +, −, *, /, (,) és \n jeleket értelmezi, ezért az **yyparse** bármilyen más karakter hatására hibajelzést ad. A 0 visszatérési érték **yyparse** számára „fájl végét" jelez.

hoc.y folytatása

```
yylex()     /* hoc1 */
{
  int c;
  while ((c=getchar()) == ' ' || c == '\t')
        ;
  if (c == EOF)
        return 0;
  if (c == '.' || isdigit(c)) { /* szám */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
  }
```

```
    if (c == '\n')
        lineno++;
    return c;
}
```

Az **yylval** változót a nyelvtani és a lexikai elemző közötti kommunikációra használjuk; ezt az **yyparse** definiálja, és típusa megegyezik a **yacc** verem típusával. Az **yylex** függvény a token típusát mint függvényértéket adja vissza, a token *értéke* (ha van) az **yylval**-ba kerül. Pl. egy lebegőpontos szám típusa **NUMBER** és az értéke mondjuk 12,34. Bizonyos tokenek esetében, különösen, ha az egyetlen jelből áll (mint a '+' és '\n'), a nyelvtan nem használja az értéket, hanem csak a típust. Ilyen esetben az **yylval**-t nem kell beállítani.

A **yacc**-nak **%token NUMBER** sora egy **#define** utasítást jelent a **yacc** kimeneti fájljában, az **y.tab.c**-ben, így a **NUMBER** a C nyelvű programban bárhol használható állandóként. A **yacc** olyan értékeket használ, amelyek nem lehetnek ASCII karakterkódok.

Ha szintaktikus hiba fordul elő, akkor az **yyparse** meghívja az **yyerror**-t, paraméterként átadva a rejtélyes "syntax error" üzenetet. A **yacc** felhasználójának kell az **yyerror**-t megírnia; a mi változatunkban egyenesen átadjuk a szöveget egy másik eljárásnak, a **warning**-nak, ami valamivel bővebb információt ír ki. A **hoc** későbbi változata a **warning**-ot közvetlenül fogja használni.

```
yyerror(s)      /* a yacc szintaktikus hibája esetében hívódik meg */
    char *s;
{
    warning(s, (char *) 0);
}
warning(s, t) /* figyelmeztetést ír ki */
    char *s, *t;
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, "%s", t);
    fprint(stderr, " near line %d\n", lineno);
}
```

S ez jelenti a **hoc.y** végét.

A **yacc** program fordítása két lépésből áll:

```
$ yacc hoc.y            A kimenetet az y.tab.c-ben hagyja.
$ cc y.tab.c -o hoc1    A végrehajtható programot hoc1-ben hagyja.
$ hoc1
2/3
    0.66666667
-3-4
hoc1: syntax error near line 1
$
```

8-1. feladat. Vizsgáljuk meg az **y.tab.c** fájl szerkezetét! (A **hoc1**-hez készült változata kb. 300 sor.) □

Kiegészítések – a mínusz előjel

Korábban azt állítottuk, hogy a **yacc** használata megkönnyíti a nyelv módosítását, fejlesztését. Példaként bővítsük ki a **hoc1**-et a mínusz előjel kezelésével, tehát úgy, hogy a

```
-3-4
```

típusú kifejezéseket is értelmezze, és ne tekintse szintaktikus hibának!

Pontosan két sorral kell bővíteni a **hoc1**-et. Egy új tokent, az UNARYMINUS-t kell a precedenciarész végére betenni:

```
%left    '+' '-'
%left    '*' '/'
%left    UNARYMINUS     /* új */
```

Ezzel a negatív előjel kapta a legnagyobb prioritást. A nyelvtan is kibővül egy újabb kifejezést előállító szabállyal:

```
expr: NUMBER      { $$ = $1; }
    | '-' expr %prec UNARYMINUS { $$ = -$2; } /* új */
```

A **%prec** azt jelenti, hogy a mínusz előjel (azaz a negatív előjel egy kifejezés előtt) precedenciaértéke az UNARYMINUS precedenciaértékével azonos (a legmagasabb); a művelet megváltoztatja az előjelet. Két kifejezés közti mínuszjel precedenciaértéke az alapértelmezés szerinti.

8-2. feladat. Bővítsük ki a **hoc1** műveletet modulus- avagy maradékképzéssel (**%**) és a pozitív előjellel (+)! Útmutatás: nézzük meg az **frexp(3)**-at! □

Egy kis kitérő a make-hez

Kellemetlen, hogy a **hoc1** egy újabb változatának lefordításához két parancsot kell leírnunk. Természetesen könnyen írható egy olyan burokfájl, ami elvégzi ezt, de van jobb megoldás is. Ez a megoldás általánosítható lesz az egynél több forrásfájlból álló programokra is. A **make** program annak a leírását olvassa be, hogy a program egyes összetevői hogyan függenek egymástól, és hogyan kell belőlük előállítani a program naprakész változatát. Ellenőrzi az összetevők utolsó változtatásának idejét és meghatározza, hogy mit kell újrafordítani a teljes új változat előállításához, majd végrehajtja az előállítást. A **make** megérti a **yacc**-hoz hasonló többlépcsős eljárások bonyodalmait is, az ilyen feladatok az egyes lépések részletezése nélkül is bevehetők a **make** leírásába.

A **make** akkor igazán hasznos, ha az előállítandó program elég nagy, s így több forrásfájlból áll össze, de a **hoc1**-hez hasonló apróságok esetén is jól jön. Íme a **hoc1**-hez készült **make**-specifikáció, amelynek egy **makefile** nevű fájlban kell lennie ahhoz, hogy a **make** feldolgozhassa:

```
$ cat makefile
hoc1:   hoc.o
        cc hoc.o -o hoc1
$
```

Ez azt jelenti, hogy a **hoc1** a **hoc.o**-tól függ, és a **hoc.o**-ból úgy lesz **hoc1**, hogy a **cc** a C fordító lefordítja, a kimenetét pedig a **hoc1**-be teszi. A **make** már tudja, hogyan kell előállítania a **hoc.y** nevű **yacc** forrásfájlból a lefordított **hoc.o** (object) fájlt:

```
$ make                  Az első teendők a hoc1 makefile alapján.
yacc hoc.y
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cc hoc.o -o hoc1
$ make                  A második nekifutás.
`hoc1' is up to date.   A make észreveszi, hogy erre már nincs szükség.
$
```

8.2. Második lépcső: változók és hibakezelés

A következő (kis) lépés a **hoc1** tárral való bővítése, és ezzel a **hoc2** előállítása. Tárként 26 változó szolgál, amelyek neve egy-egy betű **a**-tól **z**-ig. Ez nem túl elegáns megoldás, de egy kézenfekvő és hasznos közbenső lépés. Ezenkívül némi hibakezeléssel is bővíteni fogjuk programunkat. Ugyanis a **hoc1**-et próbálgatva tapasztalhatjuk, hogy a szintaktikus hibákra csak egy üzenetet ír ki, majd elhal; a nullával való osztáshoz hasonló aritmetikai hibák lekezelése pedig minden szidalmat megérdemel:

```
$ hoc1
1/0
Floating exception - core dumped
$
```

Ezen új sajátosságok érdekében keveset, kb. 35 sornyit kell változtatni programunkon. A lexikai elemzőnek, az **yylex**-nek fel kell ismernie a betűkkel jelölt változókat; a nyelvtannak tartalmaznia kell a következő formájú szabályokat:

```
expr:    VAR
       | VAR '=' expr
```

Egy kifejezés értékadást is tartalmazhat, amely többszörös értékadás is lehet:

```
x = y = z = 0
```

A változók értékeit legkönnyebben egy 26 elemű tömbben tárolhatjuk; az egy betűből álló változónév használható a tömb indexének is. Ha az elemző a változó nevét és értékét azonos veremben tárolja a feldolgozás közben, akkor a **yacc**-nek meg kell mondanunk, hogy a verme egy **double** típus helyett egy **double** és egy **int** típusú párost fogad be. Ezt a **%union** deklarációval lehet megadni a program elején. A **#define** vagy **typedef** deklaráció kiváló egy alaptípusú (pl. **double**) verem megadásához, az összetett típusokhoz azonban a **%union** mechanizmus kell, mivel a **yacc** a **$$ = $2** típusú kifejezések esetén ellenőrzi a következetes típuskezelést.

Íme a **hoc2**-nek megfelelő nyelvtani rész a **hoc.y**-ból:

```
$ cat hoc.y
%{
double mem[26];          /* tárolóhely az 'a'..'z' változóknak */
%}
%union {                 /* a verem típusa */
        double val;      /* aktuális érték */
        int    index;    /* index a mem[]-hez */
}
%token   <val>   NUMBER
%token   <index> VAR
%type    <val>   expr
%right   '='
%left    '+' '-'
%left    '*' '/'
%left    UNARYMINUS
%%
list:     /* üres */
        | list '\'
        | list expr '\n'              { printf("\t%.8g\n", $2); }
        | list error '\n'             { yyerrok; }
        ;
expr:     NUMBER
        | VAR                         { $$ = mem[$1]; }
        | VAR '=' expr                { $$ = mem[$1] = $3; }
        | expr '+' expr               { $$ = $1 + $3; }
        | expr '-' expr               { $$ = $1 - $3; }
        | expr '*' expr               { $$ = $1 * $3; }
        | expr '/' expr               {
                if ($3 == 0.0)
                                      execerror("division by zero", "");
            $$ = $1 / $3; }
        | '(' expr ')'                { $$ = $2; }
        | '-' expr %prec UNARYMINUS { $$ = -$2; }
        ;
%%                                    /* a nyelvtan vége */
...
```

A **%union** deklaráció azt mondja ki, hogy a verem egy elemében vagy egy **double** (általában egy szám) vagy egy **int** érték (a **mem** tömb egy indexe) tárolódik. A **%token** deklarációk kibővültek egy típusjelzővel. A **%type** deklaráció azt mondja meg, hogy a kifejezés az összetétel (**val**) tagja, így **double** típusú. A típus megadása teszi lehetővé a **yacc**-nak, hogy az összetétel megfelelő tagjaira hivatkozhasson. Meg kell még jegyezni, hogy az = jel jobbról asszociatív, míg a többi műveleti jel balról asszociatív.

A hibakezelés több részletből áll. A legkézenfekvőbb eset a nullával való osztás ellenőrzése; amikor az előfordul, meghívjuk az **execerror** hibarutint.

A második eset a lebegőpontos megszakítás jelzésének kezelése, amelyik akkor fordul elő, ha egy lebegőpontos szám túlcsordul. Ezt a jelet a **main** állítja be.

A hibakezelés utolsó részlete az **error**-ra vonatkozó szabályok bevezetése. Az **error** szó a **yacc** nyelvtanában egy fenntartott szó, amelynek segítségével a szintaktikus hibát kezeli. Ha hiba keletkezik, a **yacc** végül is az **error**-szabályt próbálja meg alkalmazni, vagyis a hibát nyelvtanilag „helyes"-nek tekinteni, majd folytatja az elemzést. Az **yyerrok** az elemzőben beállít egy jelzőt, ami lehetővé teszi, hogy az visszakerüljön egy működőképes elemzési állapotba. A hibakezelés minden elemzőben nehéz; nem szabad elfelejtenünk, hogy itt csak a legalapvetőbb lépésekkel foglalkoztunk, és a **yacc** lehetőségeit is csak érintettük!

A **hoc2** nyelvtanában a teendőkre vonatkozó rész nem sokat változott. A következő **main** programot kibővítettük a **setjmp**-vel, hogy a hibakezelés utáni újrakezdés érdekében megőrizzünk egy rendezett állapotot. Az **execerror** a **longjmp** megfelelője. (A **setjmp** és a **longjmp** leírása a 7.5. szakaszban található.)

```
...
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
main(argc, argv)        /* hoc2 */
        char *argv[];
{
        int fpecatch();
        progname = argv[0];
        setjmp(begin);
        signal(SIGFPE, fpecatch);
        yyparse();
}
execerror(s, t) /* futás közbeni hiba kezelése */
        char *s, *t;
{
        warning(s, t);
        longjmp(begin, 0);
}

fpecatch()      /* lebegőpontos hiba kezelése */
{
        execerror("floating print exception", (char *) 0);
}
```

Nyomkövetéshez bevált, hogy az **execerror**-ral meghívatjuk az **abort**-ot (l. **abort(3)**), ezzel kiíratjuk a tárat, amit az **adb**-vel vagy **sdb**-vel átnézhetünk. Ha a program már meglehetősen nagy, akkor az **abort**-ot a **longjmp**-vel helyettesíthetjük.

A lexikai elemzőt kicsit megváltoztattuk a **hoc2**-ben. Van egy új ellenőrzés a kisbetűkre, és mivel az **yylval** már egy összetett változó, ezért a megfelelő tagot kell beállítani az **yylex**-ből való visszatérés előtt. Íme a megváltozott részletek:

```
yylex()          /* hoc2 */
...
    if (c == '.' || isdigit(c)) { /* szám */
        ungetc(c, stdin);
        scanf("%lf", &yylval.val);
        return NUMBER;
    }
    if (islower(c)) {
        yylval.index = c - 'a'; /* csak ASCII-ra */
        return VAR;
    }
...
```

Újra figyeljük meg, hogyan különbözik a token típusa (pl. **NUMBER**) a token értékétől (pl. 3,1416).

Próbáljuk ki a **hoc2** újdonságait, a változókat és a hibakezelést!

```
$ hoc2
x = 355
        355
y = 113
        113
p = x/z                          z definiálatlan, így értéke nulla.
hoc2: division by zero near line 4     Hibakezelés!
x/y
        3.1415929
1e30 * 1e30                      Túlcsordulás!
hoc2: floating print exception near line 5
...
```

Jelenleg a PDP–11-en különleges beavatkozásokat igényel a lebegőpontos hiba észlelése, de a legtöbb gépen a **hoc2** úgy viselkedik, ahogy mutattuk.

8-3. feladat. Tegyük lehetővé az utoljára kiszámított érték megőrzését! (Ne kelljen újra begépelni, ha tovább akarunk számolni vele.) Egy lehetséges megoldás: tegyük el egy változóba, pl. az **e**-be (**e** mint előző).

8-4. feladat. Módosítsuk úgy a **hoc**-ot, hogy a kifejezés lezárójeleként, az újsor-karakterrel egyenértékűen lehessen használni a pontosvesszőt is! □

8.3. Harmadik lépcső: tetszőleges változónevek; beépített függvények

Ez a változat, a **hoc3**, több új lehetőséggel és ennek megfelelő mennyiségű programsorral bővül. A legfontosabb új lehetőség a beépített függvények elérhetősége:

```
sin   cos   atan   exp   log   log10   sqrt   int   abs
```

Bevezettünk egy hatványozási műveleti jelet (^) is; ennek van a legmagasabb prioritása és jobbról asszociatív.

Mivel a lexikai elemzőnek meg kell birkóznia az egykarakteresnél hosszabb beépített nevekkel is, különösebb erőfeszítés nélkül lehetővé tehető, hogy a változónevek tetszőleges hosszúságúak legyenek. A változók kezeléséhez bonyolultabb szimbólumtáblázatra lesz szükségünk, de ha úgyis meg kell csinálnunk, akkor előre beletehetjük néhány hasznos állandó nevét és értékét:

PI	3,14159265358979323846	π
E	2,71828182845904523536	A természetes logaritmus alapszáma
GAMMA	0,57721566490153286060	Az Euler – Mascheroni-állandó
DEG	57,29577951308232087680	Fok/radián
PHI	1,61803398874989484820	Az aranymetszés aránya

Az eredmény egy ügyes számolóprogram:

```
$ hoc3
1.5^2.3
        2.5410306
exp(2.3*log(1.5))
        2.5410306
sin(PI/2)
        1
atan(1)*DEG
        45
...
```

Mellesleg egy kicsit áttekinthetőbbé tettük a program működését is. A **hoc2**-ben az értékadás (*x = kifejezés*) nemcsak egy hozzárendelést okozott, hanem ki is írta az eredményét, hiszen minden kifejezés értéke látható:

```
$ hoc2
x = 2 * 3.14159
        6.28318          Az x változóhoz rendelt érték kiíródott.
...
```

A **hoc3** megkülönbözteti az értékadást és a kifejezést; csak a kifejezés értéke íródik ki:

```
$ hoc3
x = 2 * 3.14159          Értékadás: értéke nem íródik ki
x                        Kifejezés:
        6.28318            értéke kiíródik.
...
```

Mindezen változtatásnak eleget tevő program elég nagy (kb. 250 sor), úgyhogy a könnyebb javíthatóság és a gyorsabb fordítás érdekében különálló fájlokra tördeljük. Így öt fájlunk lesz egy helyett:

hoc.y	Nyelvtan, **main**, **yylex** (mint azelőtt).
hoc.h	Globális beépíthető adatszerkezetek.
symbol.c	Szimbólumtáblázat-kezelők: **lookup**, **install**.
init.c	Beépített függvények és állandók; **init**.
math.c	Matematikai eljárások interfésze: **Sqrt**, **Log** stb.

Az öt fájl miatt többet kell tudnunk a több fájlból álló C nyelvű programok szervezéséről, és többet kell tudnunk a **make**-ről. Van tehát bőven feladatunk!

Röviden visszatérünk a **make**-hez. Először is a szimbólumtáblázat programjára vessünk egy pillantást! A változónak van neve, típusa (a **VAR** vagy **BLTIN** típusok egyike) és értéke. Ha a változó **VAR** típusú, akkor értéke duplapontos; ha a változó **BLTIN**, akkor az értéke egy mutató, ami egy duplapontos értéket visszaadó függvényre mutat. Erre az információra a **hoc.y**, a **symbol.c** és az **init.c** részeknél lesz szükségünk, tehát háromszor kellene leírnunk. Másoláskor azonban könnyen hibázhatunk, és módosításkor is könnyű a változás átvezetéséről valahol megfeledkezni. Ezért jó a közös információkat egy fejléc- (header) fájlba, a **hoc.h**-ba helyezni, és ezt beépíteni minden olyan fájlba, amely ezeket használja. (A **.h** végződés használata csupán hagyomány, semelyik program sem követeli meg.) A **makefile**-ba is beírjuk majd a fájloknak a **hoc.h**-tól való függését, így a **hoc.h** megváltozása esetén az újrafordításokat is biztosíthatjuk.

```
$ cat hoc.h
typedef struct Symbol        /* a szimbólumtáblázat egy eleme */
        char    *name;
        short   type;        /* VAR, BLTIN, UNDEF */
        union   {
                double val; /* ha VAR típusú */
                double (*ptr)();   /* ha BLTIN */
        } u;
        struct Symbol *next;   /* másikkal való összeláncolás */
} Symbol;
Symbol *install(), *lookup();
$
```

Az **UNDEF** típus egy olyan **VAR** típus, ami még nem kapott értéket.

A szimbólumokat a **Symbol** szerkezet **next** mezőjének segítségével egy listává fűzzük össze. Ez a lista csak a **symbol.c**-ben látható; csak az **install** és a **lookup**

függvényeken keresztül lehet elérni. Ezért könnyű a szimbólumtáblázat felépítését megváltoztatni. (Már egyszer megtettük.) A **lookup** egy adott nevet keres a listában, és ha megtalálta, akkor annak mutatójával tér vissza a **Symbol**-ba, egyébként pedig nullával. A szimbólumtáblázatban lineáris keresést használunk, ez teljesen kielégítő a mi párbeszédes számolóprogramunk esetében. A változókat ugyanis csak az elemzés során kerestetjük meg, a végrehajtás során nem. Az **install** a változókat a hozzá tartozó típusával és értékével együtt a lista elejére helyezi el. Az **emalloc** a **malloc**-ot, a szabványos tárlefoglalót hívja (l. **malloc(3)**), és ellenőrzi az eredményt. Ebből a három függvényből áll a **symbol.c**. Az **y.tab.h** fájlt az **yacc −d** lefuttatása állítja elő; olyan **#define** utasításokat tartalmaz, amiket az **yacc** állít elő a **NUMBER**, **VAR** **BLTIN** stb. tokenek alapján.

```
$ cat symbol.c
#include "hoc.h"
#include "y.tab.h"
static Symbol *symlist = 0;  /* szimbólumtáblázat: láncolt lista */
Symbol *lookup(s)
        char *s;
{
        Symbol *sp;
        for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;
        return 0;       /* 0 ==> nem találja */
}
symbol *install(s, t, d)  /* s elhelyezése a szimbólumtáblázatba */
        char *s;
        int t;
        double d;
{
        Symbol *sp;
        char *emalloc();
        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1); /* +1 a '\0' miatt */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist; /* a lista elejére teszi */
        symlist = sp;
        return sp;
}
char *emalloc(n)  /* a malloc-ból való visszatérés ellenőrzése */
        unsigned n;
{
        char *p, *malloc();
        p = malloc(n);
        if (p == 0)
                execerror("out of memory", (char *) 0);
        return p;
}
$
```

Az **init.c** fájl az állandók (**PI** stb.) definícióját és a beépített függvények mutatóit tartalmazza; ezeket az **init** függvény helyezi a **main** által meghívott szimbólumtáblázatba.

```
$ cat init.c
#include "hoc.h"
#include "y.tab.h"
#include <math.h>
extern double Log(), Log10(), Exp(), Sqrt(), integer();
static struct {              /* állandók */
        char            *name;
        double          cval;
} consts[] = {
        "PI",            3.14159265358979323846,
        "E",             2.71828182845904523536,
        "GAMMA",         0.57721566490153286060,   /* Euler */
        "DEG",          57.29577951308232087680,   /* fok/radián */
        "PHI",           1.61803398874989484820,   /* az aranymetszés aránya */
        0,               0
};
static struct {                 /* beépített függvények */
        char            *name;
        double          (*func)();
} builtins[] = {
        "sin",          sin,
        "cos",          cos,
        "atan",         atan,
        "log",          Log,    /* ellenőrzi az argumentumot */
        "log10",        Log10,  /* ellenőrzi az argumentumot */
        "exp",          Exp,    /* ellenőrzi az argumentumot */
        "sqrt",         Sqrt,   /* ellenőrzi az argumentumot */
        "int",          integer,
        "abs",          fabs,
        0,              0
};
init() /* az állandók és beépített függvények feljegyzése a táblázatba */
{
        int i;
        Symbol *s;
        for (i = 0; consts[i].name; i++)
                install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
                s = install(builtins[i].name, BLTIN, 0.0);
                s->u.ptr = builtins[i].func;
        }
}
```

Az adatokat táblázatokba foglaljuk és nem keverjük a kód közé, mivel a táblázatokat könnyebb olvasni és változtatni. Táblázatainkat **static**-nek deklaráltuk, így azok csak ebben a fájlban láthatók, a program más részeiből nem. Röviden vissza fogunk még térni a **Log, Sqrt** stb. matematikai függvényekhez.

Miután az alapot felépítettük, következzék az azt használó nyelvtan megváltoztatása:

```
$ cat hoc.y
%{
#include "hoc.h"
extern double Pow().
%}
%union  {
        double val;                     /* aktuális érték */
        Symbol *sym;                    /* szimbólumtáblázat-mutató */
}
%token  <val>           NUMBER
%token  <sym>           VAR BLTIN UNDEF
%type   <val>           expr asgn
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'                     /* hatványozás */
%%
list:           /* üres */
        | list '\n'
        | list asgn '\n'
        | list expr '\n'                { printf("\t%.8g\n", $2); }
        | list error '\n'               { yyerrok; }
        ;
asgn: VAR '=' expr          { $$=$1->u.val=$3; $1->type = VAR;}
        ;
expr: NUMBER
        | VAR { if ($1->type == UNDEF)
                    execerror("undefined variable", $1->name);
                $$ = $1->u.val; }
        | asgn
        | BLTIN '(' expr')'         { $$ = (*($1->u.ptr))($3); }
        | expr '+' expr         { $$ = $1 + $3; }
        | expr '-' expr         { $$ = $1 - $3; }
        | expr '*' expr         { $$ = $1 * $3; }
        | expr '/' expr         {
                if ($3 ==0.0)
                        execerror("division by zero", "");
                $$ = $1 / $3; }
        | expr '^' expr                 { $$ = Pow($1, $3); }
        | '(' expr ')'                  { $$ = $2; }
        | '-' expr %prec UNARYMINUS { $$ = -$2; }
        ;
```

```
%%
        /* a nyelvtan vége */
...
```

Mostani nyelvtanunkban az értékadásnál már van **asgn** is és **expr** is; az olyan bemenő sor tehát, amelyben

```
VAR = expr
```

szerepel, az csak értékadás, az értéke nem íródik ki. Vegyük észre, hogy milyen egyszerűen lehetett kibővíteni a nyelvtant a hatványozással, beleértve a jobb oldali asszociativitás megadását is.

Az **yacc** vermét most egy másmilyen **%union** értelmezés adja meg: egy 26 elemű tömb indexe helyett itt egy olyan mutató áll, amely **Symbol** típusú dologra mutat. A **hoc.h** fejlécfájl tartalmazza e típus definícióját.

A lexikai elemző felismeri a változóneveket, utánanéz a szimbólumtáblázatban és eldönti, hogy ezek változók (**VAR**) vagy beépített függvények (**BLTIN**). Az **yylex** által visszaadott típusok ezek egyike lehet; a felhasználó által definiált változók és a **PI**-hez hasonlóan előredefiniált változók egyaránt **VAR** típusúak.

A változót az is jellemzi, hogy rendeltünk-e értéket hozzá vagy nem, így egy definiálatlan (amelyik nem kapott értéket) változó használatáról az **yyparse** hibajelzést adhat. A változók definiáltságának ellenőrzését a nyelvtanba kell beépíteni, ugyanis amikor egy **VAR** típust a lexikai elemző felismer, még nem ismert a szövegkörnyezete. Mi pedig nem akarunk az **x** definiálatlanságára panaszkodni, ha az adott szövegkörnyezetben ez teljesen megengedett (mint pl. egy értékadás bal oldalán az **x = 1** esetben).

Következzen az **yylex** megváltoztatott része:

```
yylex()             /* hoc3 */
...
    if (isalphat(c)) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
            *p++ = c;
        } while ((c=getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
                    s = install(sbuf, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
    }
...
```

A **main** egyetlen új sort tartalmaz; ebben hívja meg a beépített függvényeket és elhelyezi a szimbólumtáblázatba az előredefiniált változókat, pl. a **PI**-t és azokat az értékeket, amelyeket az **init** kezdetiérték-adó függvény ad.

```
main(argc, argv)                    /* hoc3 */
    char *argv[];
{
    int fpecatch();
    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}
```

Egyetlen fájl maradt még: a **math.c**. A szabványos matematikai függvények egy részéhez szükség van egy olyan hibaellenőrző interfészre, amely üzeneteket ad és lekezeli a hibát – pl. a szabványos **sqrt** függvény szó nélkül nullát ad vissza, ha argumentuma negatív. A **math.c** programjában használt hibaellenőrzések (a UNIX kézikönyv 2. fejezetében találhatók, meg l. a 7. fejezetet) sokkal megbízhatóbbak és hordozhatóbbak, mintha saját hibaellenőrzést írnánk, hiszen a függvények sajátságos korlátai feltehetően a „hivatalos" programban jobban tükröződnek. A **<math.h>** fejlécfájl tartalmazza a szabványos matematikai függvények típusdeklarációit. Az **<errno.h>** a szóbajövő hibák neveit tartalmazza.

```
$ cat matc.c
#include <math.h>
#include <errno.h>
extern int    errno;
double errcheck();
double Log(x)
        double x;
{
        return errcheck(log)x), "log");
}
double Log10(x)
        double x;
{
        return errcheck(log10(x), "log10");
}
double Exp(x)
        double x;
{
        return errcheck(exp(x), "exp");
}
double Sqrt(x)
        double x;
{
        return errcheck(sqrt(x), "sqrt");
}
double Pow(x, y)
        double x, y;
{
        return errcheck(pow(x,y), "exponentiation");
}
```

```
double integer(x)
        double x;
{
        return (double)(long)x;
}
double errcheck(d, s) /* könyvtári hívások eredményének ellenőrzése */
        double d;
        char *s;
{
        if (errno == EDOM) {
            errno = 0;
            execerror(s, "argument out of domain");
        } else if (errno == ERANGE) {
              errno = 0;
              execerror(s, "result out of range");
        }
        return d;
}
$
```

Egy érdekes (és nem nyelvtani) figyelmeztetést kapunk, ha az **yacc**-ot az új nyelvtanon futtatjuk le:

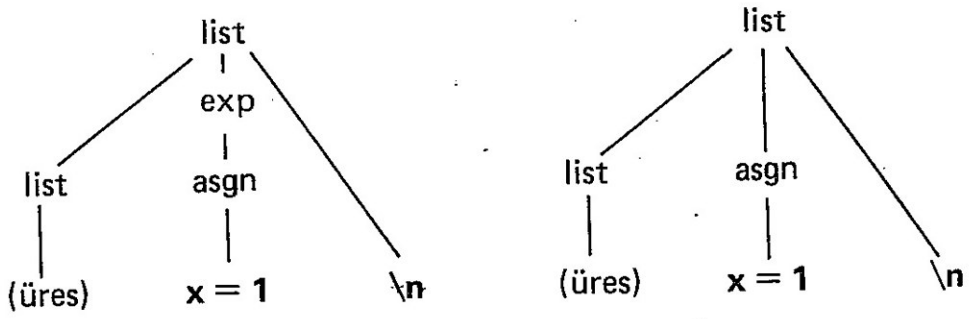
```
$ yacc hoc.y

conflicts: 1 shift/reduce
$
```

A shift/reduce üzenet azt jelenti, hogy a **hoc3** nyelvtana nem egyértelmű. Az egyetlen sorból álló

```
x = 1
```

bemenetet kétféle módon lehet értelmezni:



Az elemző úgy is dönthet, hogy az **asgn** egy **expr**-ré, majd egy **list**-té alakítható a bal oldali elemzésfa, vagy úgy is dönthet, hogy az azt követő **n**-t közvetlenül figyelembe veszi (shift), és az egészet **list**-té alakítja a közbenső szabály alkalmazása nélkül (a jobb oldali fa). Ami az ellentmondást illeti, a **yacc** a shift-et választja, mivel egy igazi

nyelvtan esetében szinte mindig az a helyes döntés. Ahhoz, hogy meggyőződhessünk a **yacc** választásának helyességéről, meg kell próbálnunk megérteni az üzeneteket.* A **−v** opcióval futó **yacc** létrehoz egy **y.output**-nak nevezett terjedelmes fájlt, ami segít kideríteni az összeütközések eredetét.

8-5. feladat. A **hoc3** szerint megengedett a következő:

```
PI = 3
```

Jó ez? Hogyan változtathatnánk meg a **hoc3**-at, hogy megtiltsa az „állandók" megváltoztatását? □

8-6. feladat. Bővítsük ki a beépített függvényeket az **atan2(x,y)**-nal, ami azt a szöget adja vissza, amelynek tangense x/y! Készítsük el a **rand()** beépített függvényt, ami egy lebegőpontos véletlenszámot ad vissza, (0, 1) intervallumbeli egyenletes eloszlással! Hogyan kell megváltoztatni a nyelvtant ahhoz, hogy a beépített függvényeknek különböző számú argumentuma lehessen? □

8-7. feladat. Hogyan lehetne elérni, hogy a **hoc** program futtatása közben parancsokat lehessen végrehajtatni, a UNIX programok **!** lehetőségéhez hasonlóan? □

8-8. feladat. Változtassuk meg a **math.c** programot úgy, hogy a lényegében azonos leírások helyett egy táblázatot használjon? □

Újabb kitérő a make-hez

Mivel a **hoc3** kódja jelenleg öt fájlban van, nem pedig egyben, a **makefile** is bonyolultabb:

```
$ cat makefile
YFLAGS = -d      #létrehozza y.tab.h-t
OBJS = hoc.o init.o math.o symbol.o   #rövidítés
hoc3:   $(OBJS)
        cc $(OBJS) -lm -o hoc3
hoc.o:  hoc.h
init.o symbol.o:    hoc.h y.tab.h
pr:
        @pr hoc.y hoc.h init.c math.c symbol.c makefile
clean:
        rm -f $(OBJS) y.tab.[ch]
$
```

* A "reduce/reduce conflicts" **yacc** üzenet súlyos problémát jelez, rendszerint nyelvtani hiba tünete, nem pedig szándékos kétértelműség.

Az YFLAGS = −d sor a **make** által előállított parancssort a **−d** opcióval bővíti; ez arra utasítja a **yacc**-et, hogy létrehozza a **#define** utasítások **y.tab.h** fájlját. Az OBJS=... sor egy, a későbbiekben gyakran használt szerkezethez egy rövidítést rendel hozzá. A szintaxis nem olyan, mint a burok változóinál volt − a zárójeleket meg kell hagyni! A **−lm** opció a matematikai könyvtárból keresi ki a matematikai függvényeket.

A **hoc3** most négy **.o** fájltól függ, egyes **.o** fájlok pedig **.h** fájloktól. Ezekre a függőségekre alapozva a **make** ki tudja következtetni, hogy mit kell újrafordítani, ha valamelyik érintett fájlban változtatunk. Ha csak látni szeretnénk, hogy mit csinál a **make**, de lefuttatni nem akarjuk, próbáljuk ki a

```
$ make −n
```

parancsot! Másrészt a fájl időparamétereit összhangba hozhatjuk a **−t** (touch : érint) opcióval anélkül, hogy valamit is újrafordítanánk.

Vegyük észre, hogy nemcsak a forrásfájlok egymástól való függéseit adtuk meg, hanem ezzel együtt egy sereg segédprogramot is. Alapértelmezés szerint a **make** először a **makefile** listájában legelöl álló teendőt hajtja végre, de ha az egyik függőségi szabály címkéjére hivatkozunk, pl. a **symbol.o**-ra vagy a **pr**-re, akkor azt fogja végrehajtani helyette. Egy üres függőségi szabály azt jelenti, hogy ez az egység sosem „naprakész", tehát ez a teendő minden híváskor elvégezendő. Így a

```
$ make pr / lpr
```

kiírja az általunk kért listát egy sornyomtatón. A vezető @ (a **@pr**-ben) letiltja a **make** által éppen végrehajtott parancs visszhangozását. Végül a

```
$ make clean
```

törli a **yacc** kimeneti fájlait és a **.o** fájlokat.

A **makefile**-nak az üresszabály-mechanizmusa gyakran alkalmasabb arra, hogy az összes összekapcsolódó teendőt egyetlen fájlban helyezzük el, mint a burokfájlok. Ráadásul a **make**-et nem kell a programfejlesztésre korlátozni − jól használható időfüggő tevékenységek tetszőleges csoportjának összehangolásához.

Kitérő a lex-hez

A **lex** program lexikai elemzőket állít elő. Működése hasonló a szintaktikus elemzőket előállító **yacc** működéséhez: szabályos kifejezésekkel leírjuk nyelvünk lexikai szabályainak specifikációját, és az ezekkel egyező karakterláncok esetén végrehajtandó C nyelvű programrészleteket. Ebből a **lex** egy felismerőprogramot készít. A **lex** és a **yacc** ugyanazon a mechanizmuson keresztül támogatja egymást, mint az általunk korábban írt lexikai elemzők. Nem áll szándékunkban itt a **lex** részleteibe bocsátkozni, a következőkben csupán az a célunk, hogy kedvet csináljunk a további tanuláshoz. Tekintsünk bele a **lex** referenciakönyvbe (megtalálható a UNIX Programozási kézikönyv 2B kötetében)!

Először is lássuk a **lex** programot, ami a **lex.l** fájlban található! Ez váltja fel az eddig használt **yylex** függvényt.

```
$ cat lex.l
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t] { ; }   /* szóköz és tabulátor átlépése */
[0-9]+\.?|[0-9]*\.[0-9]+ {
    sscanf(yytext, "%lf", &yylval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
    Symbol *s;
    if ((s=lookup(yytext)) == 0)
            s = install(yytext, UNDEF, 0.0);
    yylval.sym = s;
    return s->type == UNDEF ? VAR : s->type; }
\n  { lineno++; return '\n'; }   /* minden egyéb */
.   { return yytext[0]; }
$
```

Minden szabály olyan szabályos kifejezés, mint amilyen az **egrep**-ben és **awk**-ban is szerepelt. A **lex** azonban olyan C-szerű **escape** jeleket is felismer, mint a **\t** és **\n**-t. A teendőket kapcsos zárójel zárja közre. A szabályokat sorrendjükben próbálja ki, a * és a + jelekhez hasonló szerkezetek a lehető leghosszabb szövegrészhez illeszkednek. Ha a szabály illeszkedik a bemeneten levő részhez, akkor a teendőket a **lex** végrehajtja. Az illeszkedő bemenő lánc megtalálható az **yytext**-nek nevezett **lex**-láncban.

A **make** fájlt a **lex** használatához meg kell változtatni:

```
$ cat makefile
YFLAGS = -d
OBJS = hoc.o lex.o init.o math.o symbol.o
hoc3: $(OBJS)
      cc $(OBJS) -lm -ll -o hoc3
hoc.o: hoc.h
lex.o init.o symbol.o:     hoc.h y.tab.h
...
$
```

A **make** most is tudja, hogy a **.l** fájlból hogyan kell előállítania a megfelelő **.o** fájlt, ehhez nekünk csak azt kell megmondani, hogy mi mitől függ. (A **cc** által végignézendő listában a **-ll** nevű **lex**-könyvtárat is meg kell adnunk, mivel a **lex** által előállított felismerő ezt is használja.) A látványos eredmény teljesen automatikusan áll elő:

```
$ make
yacc -d hoc.y
conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
lex lex.l
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc -c init.c
cc -c math.c
cc -c symbol.c
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -o hoc3
$
```

Ha egy fájl megváltozott, akkor egyetlen **make** parancs elegendő egy naprakész változat előállításához:

```
$ touch lex.l      Megváltoztatja lex.l módosításának idejét.
$ make
lex lex.l
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc hoc.o lex.o init.o math.o symbol.o -ll -lm -o hoc3
$
```

Sokáig tanakodtunk arról, hogyan tárgyaljuk a **lex**-et. Csupán kitérőként kezeljük, rövid bemutatás után félretéve, vagy részletesen mint a bonyolult nyelvek lexikai elemzésének alapvető eszközét. Mindkettő mellett szólnak érvek. A **lex** legnagyobb hibája (attól eltekintve, hogy használójának még egy új nyelvet meg kell tanulnia), hogy lassan fut és lassabb, terjedelmesebb elemzőket állít elő, mint a megfelelő C nyelvű változat. Ráadásul a nem szokványos megoldásoknál nehezebb a bemeneti rendszerét illeszteni, mint pl. a hibakezelést vagy a bemeneti fájlt. Ezek egyike sem jelent komoly gondot a **hoc** esetében. A fő korlát a hely: több oldal kell a **lex**-változat leírásához, így a további lexikai elemzéseknél (vérző szívvel) vissza fogunk térni a C nyelvhez! Ennek ellenére a **lex**-változat elkészítése jó gyakorlatnak számít.

8-9. feladat. Hasonlítsuk össze a **hoc3** két változatának méretét! L. **size(1)**. □

8.4. Negyedik lépcső: géprevitel

A **hoc5** irányába haladunk tovább. Ez már egy vezérlő utasításokkal is ellátott nyelv értelmezője lesz. A **hoc4** egy közbeeső lépés, a **hoc3**-mal azonos szintű szolgáltatásokat nyújt, de már a **hoc5** értelmezőjével valósítjuk meg. A **hoc4**-et azért írtuk meg, mert

így két azonosan viselkedő programunk lesz, és ez a nyomkövetésnél jól használható. A bemenet elemzésekor a **hoc4** egy egyszerű számítógép számára állít elő programot ahelyett, hogy azonnal kiszámítaná a választ. Az utasítás végének elérésekor az előállított kódot végrehajtja (értelmezi), ezzel kiszámítja a kívánt eredményt.

Ez az egyszerű számítógép egy *veremgép:* ha talál egy operandust, azt a verembe teszi (pontosabban, kódot generál arra, hogy a verembe tegye); a legtöbb művelet a verem tetején levő elemekkel dolgozik. Pl. az

```
x = 2 * y
```

értékadás kezelésére a következő kódot állítja elő:

```
constpush   A verembe tesz egy állandót
   2          ...a 2 állandót.
varpush     A szimbólumtáblázat mutatóját teszi a verembe
   y          ... az y változó miatt.
eval        Kiszámítás: a mutatót értékre cseréli.
mul         A két felső elem összeszorzása, helyükre az eredmény kerül.
varpush     Verembe teszi a szimbólumtáblázat mutatóját
   x          ... az x változó miatt.
assign      Az érték a változóba kerül, a mutatót leveszi.
pop         Kiveszi a felső értéket a veremből.
STOP        Vége az utasítássorozatnak.
```

A program a kifejezést kiértékeli és az eredményt az **x**-be teszi, ahogy ezt a megjegyzések is mutatják. Az utolsó **pop** utasítás eltávolítja az eredményt a veremből, mivel arra ott már nincs szükség.

A veremgépek általában egyszerű értelmezők. A miénk sem kivétel – ez csak egy tömb, amelyben operátorok és operandusok vannak. Az operátorok a gép utasításai; mindegyik egy függvényhívás, amelynek argumentumai – ha vannak – követik az utasítást. Más operandusok már előre a veremben lehetnek, mint ahogyan az előző példában is láttuk.

A **hoc4** szimbólumtáblázata azonos a **hoc3**-éval; az **init.c**-beli kezdetiérték-adás és a **math.c**-beli matematikai függvények is ugyanazok. A nyelvtan is azonos a **hoc3**-éval, de a teendők teljesen eltérőek. Lényegében minden tevékenység gépi utasításokat és hozzájuk tartozó argumentumokat állít elő. Pl. egy kifejezésbeli **VAR**-hoz három tételt állít elő: egy **varpush** utasítást, a változó szimbólumtáblázatbeli mutatóját és egy **eval** utasítást, ami végrehajtásakor kicseréli a szimbólumtáblázatbeli mutatót az értékére. A *-nak megfelelő program egyszerűen csak a **mul** utasítás, mivel operandusai már a veremben vannak.

```
$ cat hoc.y
%{
#include "hoc.h"
#define code2(c1,c2)     code(c1); code(c2)
#define code3(c1,c2,c3)  code(c1); code(c2); code(c3)
%}
%union {
Symbol *sym;          /* szimbólumtáblázat-mutató */
```

```
        Inst    *inst;          /* gépi utasítás */
}
%token  <sym> NUMBER VAR BLTIN UNDEF
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS
%right  '^'                     /* hatványozás */
%%
list:                           /* üres */
        | list '\n'
        | list asgn '\n'    { code2(pop, STOP); return 1; }
        | list expr '\n'    { code2(print, STOP); return 1; }
        | list error '\n'   { yyerrok; }
        ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); }
        ;
expr:     NUMBER        { code2(constpush, (Inst)$1); }
        | VAR           { code3(varpush, (Inst)$1, eval); }
        | asgn
        | BLTIN '(' expr ')' { code2(bltin, (Inst)$1->u.ptr); }
        | '(' expr ')'
        | expr '+' expr  { code(add); }
        | expr '-' expr  { code(sub); }
        | expr '*' expr  { code(mul); }
        | expr '/' expr  { code(div); }
        | expr '^' expr  { code(power); }
        | '-' expr %prec UNARYMINUS { code(negate): }
        ;
%%                          /* a nyelvtan vége */
...
```

Az **Inst** a gépi utasítás adattípusa (mutató egy **int**-et visszaadó függvényre), erre hamarosan visszatérünk. Megjegyzendő, hogy a **code** argumentumai függvénynevek, azaz függvénynevek mutatói vagy más, függvénymutatót jelentő értékek.

A **main**-t is valamelyest megváltoztattuk. Az elemző most minden utasítás vagy kifejezés után visszatér, és az előállított programot végrehajtja. Az **yyparse** a fájl vége érzékelésekor nullát ad vissza.

```
main(argc, argv)                /* hoc4 */
    char *argv[];
{
    int fpecatch();
    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); yyparse(); initcode())
        execute(prog);
    return 0;
}
```

A lexikai elemző csak nagyon kicsit változott. A fő változás az, hogy a számokat meg kell őrizni, nem pedig azonnal felhasználni. A legegyszerűbben ezt úgy érhetjük el, hogy a változókkal együtt ezeket is a szimbólumtáblázatba helyezzük el. Az **yylex** megváltoztatott része a következő:

```
yylex()                          /* hoc4 */
...
    if (c == '.' || isdigit(c)) { /* szám */
      double d;
      ungetc(c, stdin);
      scanf("%lf", &d);
      yylval.sym = install("", NUMBER, d);
      return NUMBER;
    }
...
```

Az értelmező vermében levő elemek vagy lebegőpontos értékek, vagy a szimbólumtáblázat elemeinek mutatói; a verem adattípusa ezekből összetett típus. Maga a gép pedig egy mutatókat tartalmazó tömb, a mutatók vagy egy műveletet végrehajtó rutinra (mint pl. a **mul**), vagy a szimbólumtáblázat egy adatára mutatnak. A **hoc.h** fejlécfájlt tehát ki kell bővítenünk ezekkel az adatszerkezetekkel és az értelmezőnek szóló függvénydefiníciókkal. Így ezek a programban mindenütt ismertek lesznek. (Közbevetőleg megjegyezzük: úgy döntöttünk, hogy az összes információt egy fájlban helyezzük el, nem pedig többen. Nagyobb programok esetében érdemes lehet több fájlba tenni a fejlécinformációkat, és mindegyiket csak oda építeni be, ahol valóban szükségesek.)

```
$ cat hoc.h
typedef struct   Symbol  { /* a szimbólumtáblázat eleme */
        char     *name;
        short    type;     /* VAR, BLTIN, UNDEF */
        union {
                 double val;          /* ha VAR */
                 double (*ptr)();     /* ha BLTIN */
        } u;
        struct Symbol *next;  /* láncolás a másikhoz */
} Symbol;
Symbol *install(), *lookup();
typedef union Datum { /* az értelmező vermének típusa */
        double val;
        Symbol *sym;
} Datum;
extern Datum pop();
typedef int (*Inst)();  /* gépi utasítás */
#define STOP (Inst) 0
extern Inst prog[];
extern eval(), add(), sub(), mul(), div(), negate(), power();
extern assign(), bltin(), varpush(), constpush(), print();
$
```

Azokat a rutinokat, amelyek végrehajtják a gép utasításait és a vermet kezelik, a **code.c** nevű fájlba tettük. Ez kb. 150 sor hosszú, ezért részenként fogjuk bemutatni.

```
$ cat code.c
#include "hoc.h"
#include "y.tab.h"
#define  NSTACK 256
static   Datum stack[NSTACK];   /* a verem */
static   Datum *stackp;         /* a verem következő szabad helye */
#define  NPROG 2000
Inst     prog[NPROG];           /* a gép */
Inst     *progp;                /* kódgeneráláshoz az első szabad hely */
Inst     *pc;                   /* programszámláló a végrehajtáskor */
initcode()                      /* kódgenerálás előkészítése */
{
        stackp = stack;
        progp = prog;
}
...
```

A vermet **pop-** és **push**-hívásokkal kezeljük:

```
push(d)         /* d-t a veremre helyezi */
        Datum d;
{
        if (stackp >= &stack[NSTACK])
                execerror("stack overflow", (char *) 0);
        *stackp++ = d;
}
Datum pop()     /* előveszi a verem felső elemét */
{
        if (stackp <= stack)
                execerror("stack underflow", (char *) 0);
        return *--stackp;
}
```

A gépet az elemzés során a **code** függvény ismételt hívásával állítjuk elő. A hívások egyszerűen egy-egy utasítást tesznek a **prog** tömb első szabad helyére. Visszakapjuk az utasítás helyét is, de azt a **hoc4**-ben nem használjuk fel.

```
Inst *code(f)     /* egy utasítás vagy operandus elhelyezése */
        Inst f;
{
        Inst *oprogp = progp;
        if (progp >= &prog[NPROG])
                execerror("program too big", (char *) 0);
        *progp++ = f;
        return oprogp;
}
```

A gép működtetése egyszerű; ha már készen van, akkor meglepően kicsi az a rutin, ami „futtatja" a gépet:

```
execute(p)        /* a gép működtetője */
        Inst *p;
{
        for (pc = p; *pc != STOP; )
                (*(*pc++))();
}
```

Minden ciklus végrehajtja azt a függvényt, amelyikre a **pc** programszámláló által megadott utasítás mutat, majd eggyel növeli a **pc** értéket, így készenáll a következő utasítás végrehajtására. A **STOP** utasítás befejezteti a ciklust. Bizonyos utasítások, mint pl. a **constpush** és a **varpush**, szintén növelik a **pc**-t, hogy átlépődjenek az utasítást követő argumentumok.

```
constpush()     /* állandó lerakása a veremre */
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()     /* változó lerakása a veremre */
{
    Datum d;
    d.sym = (Symbol *)(*pc++);
    push(d);
}
```

A gép hátralevő része egyszerű. Pl. az aritmetikai műveletek lényegében azonosak: egyetlen alaptípusból keletkeztek. Íme pl. az **add**:

```
add()       /* a verem két felső elemét összeadja */
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}
```

A hátralevő rutinok ugyanilyen egyszerűek.

```
eval()       /* a veremben levő változó kiszámítása */
{
    Datum d;
    d = pop();
```

```
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}
assign() /* a legfelső elem értékét a következő elemhez rendeli */
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym-> type != VAR && d1.sym->type != UNDEF)
       execerror("assigment to non-variable",
            d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}
print()   /* kiveszi a verem legfelső elemét és kiírja */
{
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}
bltin()   /* a veremen levő beépített függvény kiszámítása */
{
    Datum d;
    d = pop();
    d.val = (*(double (*)())(*pc++))(d.val);
    push(d);
}
```

Legnehezebb megérteni a **bltin** leírását, ez azt mondja, hogy ***pc**-nek egy dupla típussal visszatérő függvény mutatójára kell állnia és ezt a függvényt **d.val** argumentummal kell végrehajtani.

Az **eval** és az **assign** hibaüzenetei sohasem fordulhatnak elő, ha minden rendben működik; arra az esetre hagytuk meg ezeket, ha egyes programhibák tönkretennék a vermet. Ez időben és helyben nem kerül annyiba, mint amennyit megér, hogy felfedezzük a hibát, amit egy gondatlan változtatással okoztunk. (Nekünk is jól jött, többször is.)

Az, hogy a C nyelv képes függvénymutatókat kezelni, tömör és hatékony programot eredményez. Egy másik lehetséges megközelítés, amelynek megvalósítása gyakorló feladatnak marad: az operátorokat állandóknak tekintjük, és a szemantikai függvényeket az **execute** rutinban egy hatalmas **switch** utasítássá építjük össze.

Harmadik kitérő a make-hez

A **hoc** forráslistájának növekedésével egyre hasznosabbá válik a programban annak mechanikus nyomonkövetése, hogy mi változott és mi függ a változásoktól. A **make**

szépsége abban áll, hogy automatizálja azt a munkát, amit egyébként kézzel végeznénk el (és néha elrontanánk) vagy burokfájllal oldanánk meg.

Két dologban javítottuk a **makefile**-t. Az első azon a tapasztalaton alapszik, hogy bár több fájl függ az **y.tab.h**-ban levő és a **yacc** által definiált állandóktól, ha az állandók nem változnak meg, akkor nincs szükség az újrafordításra – a **hoc.y** C nyelvű programjának a változtatása nincs semmi másra hatással. Az új **makefile**-ban a **.o** fájlok egy új **x.tab.h** fájltól függenek, ami csak akkor változik, ha az **y.tab.h** összetétele változik. A második javítás az, hogy a **pr**-be (a forrásfájlok kiírójába) beépítettük, hogy függjön a forrásfájloktól: így csak a megváltozott fájlok íródnak ki.

Az első változtatás nagyobb programok esetén sok időt takarít meg, ha a nyelvtan nem változik, de a szemantika igen (általában ez a jellemző). A második változtatással sok-sok papírt takaríthatunk meg.

Íme a **hoc4** új **makefile**-ja:

```
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o
hoc4:   $(OBJS)
        cc $(OBJS) -lm -o hoc4
hoc.o code.o init.o symbol.o: hoc.h
code.o init.o symbol.o: x.tab.h
x.tab.h: y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
pr:     hoc.y hoc.h code.c init.c math.c symbol.c
        @pr $?
        @touch pr
clean:
        rm -f $(OBJS) [xy].tab.[ch]
```

A **cmp** előtti mínuszjel (–) azt jelenti a **make** számára, hogy dolgozzon tovább akkor is, ha a **cmp** hibát érzékel; ez teszi lehetővé, hogy az **x.tab.h** hiánya esetén is működjön az eljárás. (A **-s** opció következtében a **cmp** nem készít kimenetet, de a kilépési státust beállítja.) A $? szimbólum kifejtésekor a szabály azon elemeinek listáját kapjuk meg, amelyek nem naprakészek. Sajnálatos módon a **make** jelzésrendszere legjobb esetben is csak lazán kapcsolódik a burokéhoz.

Most bemutatjuk ezek működését. Tegyük fel, hogy minden naprakész állapotban van. Ekkor:

```
$ touch hoc.y          A hoc.y idejét megváltoztatjuk.
$ make
yacc -d hoc.y

conflicts: 1 shift/reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
```

```
cc hoc.o code.o init.o math.o symbol.o -lm -o hoc4
$ make -n pr          A megváltozott fájlok kiírása.
pr hoc.y
touch pr
$
```

Vegyük észre, hogy a **hoc.y** kivételével semmi sem került újrafordításra, mivel az **y.tab.h** fájl nem változott.

8-10. feladat. Tegyük a **verem** és **prog** méretét dinamikussá úgy, hogy a **hoc4** soha ne merítse ki a tárat, amíg **malloc**-hívással lehet területet lefoglalni! □

8-11. feladat. Módosítsuk a **hoc4**-et úgy, hogy az **execute**-ban a műveletek típusa szerint a függvényhívások helyett egy **switch**-et használjunk! Hogyan vethetők össze e változatok a forrássorok és a végrehajtási sebesség szempontjából? Karbantartás és továbbfejlesztés szempontjából milyen eltérések várhatók? □

8.5. Ötödik lépcső: programvezérlés; relációs operátorok

Ez a változat, a **hoc5** aratja le annak a munkának a gyümölcsét, amit az értelmező kifejlesztésébe fektettünk. A **hoc5** lehetővé teszi az **if-else** és **while** utasításoknak a C nyelvhez hasonló használatát, az utasítások csoportosítását, kapcsos zárójelek, { és } segítségével, valamint elfogad egy **print** utasítást. Ebbe a változatba már beépítettük a relációs operátorok teljes készletét (>, > = stb.), akárcsak az AND és az OR műveleteket (&&, | |). (Ez utóbbi kettő nem biztosítja a balról jobbra való kiértékelést, ami a C nyelv nagy erőssége; mindkét feltétel kiértékelődik akkor is, ha ez nem lenne szükséges.)

A nyelvtan kibővült a tokenekkel, a nemterminálisokkal, valamint az **if**, **while**, kapcsos zárójelek és a relációs operátorok szabályaival. Ez jócskán megnövelte a programot, de (talán az **if** és a **while** kivétel) nem tette sokkal bonyolultabbá.

```
$ cat hoc.y
%{
#include "hoc.h"
#define     code2(c1,c2)     code(c1); code(c2)
#define     code3(c1,c2,c3) code(c1); code2(c2); code3(c3)
%}
%union {
            Symbol *sym;      /* szimbólumtáblázat-mutatója */
            Inst    *inst;    /* gépi utasítás */
}
%token      <sym> NUMBER PRINT VAR BLTIN UNDEF WHILE IF ELSE
%type       <inst> stmt asgn expr stmtlist cond while if end
```

```
%right      '='
%left       OR
%left       AND
%left       GT GE LT LE EQ NE
%left       '+' '-'
%left       '*' '/'
%left       UNARYMINUS NOT
%right      '^'
%%
list:         /* üres */
            | list '\'
            | list asgn '\n'        { code2(pop, STOP); return 1; }
            | list stmt '\n'        { code(STOP); return 1; }
            | list expr '\n'        { code2(print, STOP); return 1; }
            | list error '\n'       { yyerrok; }
            ;
asgn:         VAR '=' expr { $$=$3; code3(varpush, (Inst)$1, assign); }
            ;
stmt:         expr          { code(pop); }
            | PRINT expr { code(prexpr); $$ = $2; }
            | while cond stmt end {
                  ($1)[1] = (Inst)$3;     /* ciklusmag */
                  ($1)[2] = (Inst)$4; }   /* vége, ha a feltétel hamis */
            | if cond stmt end { /* else nélküli if */
                ($1)[1] = (Inst)$3;     /* then rész */
                ($1)[3] = (Inst)$4; }   /* vége, ha a feltétel hamis */
            | if cond stmt end ELSE stmt end { /* if és else */
                ($1)[1] = (Inst)$3;     /* then rész */
                ($1)[2] = (Inst)$6;     /* else rész */
                ($1)[3] = (Inst)$7; }   /* vége, ha a feltétel hamis */
            | '{' stmtlist '}' { $$ = $2; }
            ;
cond:         '(' expr ')' { code(STOP); $$ = $1; }
            ;
while:      WHILE { $$ = code3(whilecode, STOP, STOP); }
            ;
if:           IF { $$=code(ifcode); code3(STOP, STOP, STOP); }
end:          /* üres */ { code(STOP); $$ = progp; }
            ;
stmtlist:     /* üres */ { $$ = progp; }
            | stmtlist '\n'
            | stmtlist stmt
            ;
expr:         NUMBER { $$ = code2(constpush, (Inst)$1); }
            | VAR       { $$ = code3(varpush, (Inst)$1, eval); }
            | asgn
            | BLTIN '(' expr ')'
                { $$ = $3; code2(bltin, (Inst)$1->u.ptr); }
            | '(' expr ')'        { $$ = $2; }
            | expr '+' expr   { code(add); }
            | expr '-' expr   { code(sub); }
```

```
        | expr '*' expr    { code(mul);}
        | expr '/' expr    { code(div); }
        | expr '^' expr    { code(power); }
        | '-' expr %prec UNARYMINUS { $$ = $2; code(negate); }
        | expr GT expr     { code(gt); }
        | expr GE expr     { code(ge); }
        | expr LT expr     { code(lt); }
        | expr LE expr     { code(le); }
        | expr EQ expr     { code(eq); }
        | expr NE expr     { code(ne); }
        | expr AND expr    { code(and); }
        | expr OR expr     { code(or); }
        | NOT expr         { $$ = $2; code(not); }
        ;
%%
```

A nyelvtanban öt shift/reduce ütközés van, mindegyik a **hoc3**-ban említetthez hasonló.

Vegyük észre, hogy most több helyen zárja le **STOP** utasítás a sorozatokat; mint korábban, most is a **progp** az a hely, ahol létrejön a következő utasítás. A **STOP** utasítások végrehajtásával lesz vége az **execute**-beli ciklusnak. Az **end**-re vonatkozó szabály valójában egy olyan szubrutin, amely bárhonnan híva **STOP** utasítást generál, és az őt követő utasítás címével tér vissza.

A **while** és az **if** számára előállított kód részletes tanulmányozást igényel. Amikor a **while** kulcsszóra bukkanunk, a **whilecode** nevű eljárást hozzuk létre, ennek gépbeli helyét a

```
while: WHILE
```

szabály értéke adja meg. Ezzel egyidőben a gép következő két helyét is lefoglaljuk, de ezeket csak később töltjük ki. Az ezután létrehozott kód a **while** feltételrészét alkotó kifejezés lesz. A **cond** által visszaadott érték a feltétel kódjának a kezdetére mutat. A teljes **while** utasítás felismerése után, a **whilecode** utasítás mögött fenntartott két helyre a ciklusmag és a ciklusmagot követő utasítás címét írjuk. Ennek az utasításnak a kódját a következő lépésben állítjuk majd elő.

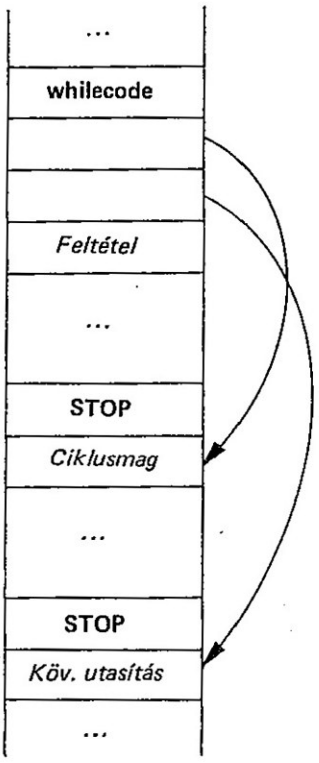
```
| while cond stmt end {
        ($1)[1] = (Inst)$3;     /* ciklusmag */
        ($1)[2] = (Inst)$4; }   /* vége, ha a feltétel hamis */
```

A **$1** a **whilecode** tárbeli címe, **($1)[1]** és **($1)[2]** pedig az ezt követő két cím.

A következő oldalon levő ábra segítségével világosabb lesz az elrendezés.

Az **if** esetében hasonló a helyzet, de ott három helyet tartunk fenn, a **then** és az **else** részeknek, valamint az **if**-et követő utasításnak. Röviden vissza fogunk térni ennek végrehajtására.

A lexikai elemzés most valamivel hosszabb, lényegében kiegészült az újabb műveleti jelek leírásával.




```
yylex()        /* hoc5 */
...
        switch (c) {
        case '>':       return follow('=', GE, GT);
        case '<':       return follow('=', LE, LT);
        case '=':       return follow('=', EQ, '=');
        case '!':       return follow('=', NE, NOT);
        case '|':       return follow('|', OR, '|');
        case '&':       return follow('&', AND, '&');
        case '\n':      lineno++; return '\n';
        default:        return c;
        }
}
```

A **follow** egy karakterrel előbbre néz, és ha nem az, amit várt, akkor az **ungetc** segítségével visszateszi a bemenetre.

```
follow(expect, ifyes, ifno)   /* >= stb. azonosítására */
{
        int c = getchar();
        if (c == expect)
            return ifyes;
        ungetc(c, stdin);
        return ifno;

}
```

A **hoc.h**-ba további függvénydeklarációk kerültek – pl. az összes reláció –, de a lényeg egyébként ugyanaz, mint a **hoc4**-ben. Íme az utolsó pár sor:

```
$ cat hoc.h
...
typedef int (*Inst)();   /* gépi utasítás */
#define STOP (Inst) 0
extern Inst prog[], *progp, *code();
extern eval(), add(), sub(), mul(), div(), negate(), power();
extern assign(), bltin(), varpush(), constpush(), print();
extern prexpr();
extern gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern ifcode(), whilecode();
$
```

A **code.c** nagyobb része szintén ugyanaz, mint korábban, bár van egy csomó magától érthetően egyszerű új eljárás a relációs operátorok kezelésére. Jellemző példaként íme az **le** („less than or equal to", kisebb vagy egyenlő) függvény:

```
le()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val <= d2.val);
        push(d1);
}
```

A két, nem nyilvánvalóan érthető rutin a **whilecode** és az **ifcode**. Megértésük azon múlik, hogy észrevegyük: az **execute** addig hajtja végre az utasítások sorozatát, amíg egy **STOP** utasítást talál, ekkor visszatér. Az elemzés során a kódot gondosan úgy hoztuk létre, hogy minden olyan utasítássorozatot, amelyet egyetlen **execute**-hívásnak kell végrehajtania, egy **STOP** zárjon le. A **while** ciklusmagját és a hozzá tartozó feltételt, az **if**-nek a **then** és **else** részét egyaránt ismétlődő **execute** hívások hajtják végre. Amikor befejezik működésüket, az eredeti szintre kell visszatérniük. Ezen ismétlődő feladatok vezérlését a **whilecode** és az **ifcode** programjai végzik, amelyek közvetlenül megfelelnek a **while** és az **if** utasításoknak.

```
whilecode()
{
        Datum d;
        Inst *savepc = pc;          /* ciklusmagra mutat */
        execute(savepc+2);          /* feltétel */
        d = pop();
        while (d.val) {
                execute(*((Inst **)(savepc)));  /* ciklusmag */
                execute(savepc+2);
                d = pop();
        }
        pc = *((Inst **)(savepc+1));  /* következő utasítás */
}
```

Idézzük fel korábbi elemzésünkből, hogy a **whilecode** műveletet egy mutató követi, amely a ciklusmagra mutat, ezt az utána álló utasítás mutatója, végül a feltételrész elejét kijelölő mutató követi! A **whilecode** hívása előtt a **pc**-t (az utasításszámlálót) már átállítjuk, így az már a ciklusmag mutatójára mutat. Így a **pc+1** a következő utasításra és a **pc+2** feltételre mutat.

Az **ifcode** nagyon hasonló ehhez: belépéskor a **pc** a **then** részre, a **pc+1** az **else** részre, a **pc+2** a következő utasításra, a **pc+3** pedig a feltételrészre mutat.

```
ifcode()
{
        Datum d;
        Inst *savepc = pc; /* then rész */
        execute(savepc+3); /* feltétel */
        d = pop();
        if (d.val)
            execute(*((Inst **)(savepc)));
        else if (*((Inst **(savepc+1)))   /* else rész? */
            execute(*((Inst **)(savepc+1)));
        pc = *((Inst **)(savepc+2));   /* következő utasítás */
}
```

Az **init.c** kezdetiérték-adás programja is kibővült egy kicsit, nevezetesen a kulcsszavak táblázatával, amit minden mással együtt a szimbólumtáblázatban tárolunk.

```
$ cat init.c
...
static struct {        /* kulcsszavak */
        char *name;
        int     kval;
} keywords[] = {
        "if",                   IF,
        "else",                 ELSE,
        "while",                WHILE,
        "print",                PRINT,
        0,                      0,
};
...
```

Az **init**-be kell írnunk egy újabb ciklust a kulcsszavak táblázatba helyezéséhez.

```
for (i = 0; keywords[i].name; i++)
      install(keywords[i].name, keywords[i].kval, 0.0);
```

A szimbólumtáblázat kezelésén semmit sem kell változtatni; a **code.c**-t kiegészítjük a **prexpr** rutinnal, amelyet egy **print** kifejezés alakú utasítás végrehajtása során hívunk meg.

```
prexpr()    /* numerikus érték kiírása */
{
        Datum d;
        d = pop();
        printf("%.8g\n", d.val);
}
```

Ez nem a **print** függvény; az a verem tetejéről levesz egy elemet, és egy tabulátorkaraktert tesz a kimenethez.

A **hoc5** már egy egészen hasznos számológéppé vált, bár a komoly programozás még több lehetőséget igényelne. A következő feladatok sugalmaznak néhányat.

8-12. feladat. Változtassuk meg a **hoc5**-öt úgy, hogy nyomkövetés céljából olvasható formában írja ki az általa létrehozott gépet! □

8-13. feladat. Vezessük be a C nyelvben található értékadási műveleteket (mint pl. a $+=$, $*=$ stb.), valamint a növelő és a csökkentő műveletet ($++$ és $--$)! Változtassuk meg az && és a || műveletet úgy, hogy biztos legyen a balról jobbra történő kiértékelés, és ne értékelődjön ki fölöslegesen a jobb oldal (ahogyan ez a C nyelvben is van)! □

8-14. feladat. Vezessünk be a **hoc5**-be egy C-szerű **for** utasítást, valamint **break**-et és **continue**-t! □

8-15. feladat. Hogyan tudnánk megváltoztatni a **hoc5** nyelvtanát vagy lexikai elemzőjét (vagy mindkettőt) úgy, hogy elnézőbb legyen az újsor-karakterek elhelyezésével? Hogyan érhető el, hogy a pontosvessző ugyanazt a szerepet töltse be, mint az újsor-karakter? Hogyan lehetne magyarázószövegeket is használni a programban? Milyen szintaktikát használnánk ezekhez? □

8-16. feladat. Bővítsük ki a **hoc5**-öt megszakításkezeléssel, hogy egy elszabadult számolást le lehessen állítani anélkül, hogy elvesztenénk a már kiszámított változók értékét! □

8-17. feladat. Bosszantó, hogy egy programot előbb egy fájlban létre kell hoznunk, majd futtatnunk, és aztán egy apró változtatás eléréséért újra kell kezdenünk az egészet. Hogyan tudnánk megváltoztatni a **hoc5**-öt, hogy megengedjen egy **edit** (javítási) parancsot, ami azt eredményezné, hogy átlépnénk egy szerkesztőprogramba, amelyben már benne is van **hoc** programunk egy másolata? Útmutatás: tekintsük a **text** műveleti kódját! □

8.6. Hatodik lépcső: függvények és eljárások; be- és kimenet

A **hoc** fejlesztésének – legalábbis ebben a könyvben – az utolsó lépéséhez érkeztünk. Jelentős bővítésként függvényekkel és eljárásokkal egészítjük ki. Lehetővé tesszük, hogy szövegeket is kiírhassunk, ne csak számokat, és hogy a szabványos bemenetről értékeket olvashassunk be. A **hoc6** fájlnév argumentumokat is elfogad, beleértve a szabványos bemenet – nevét is. E változások együttesen 235 új programsort jelentenek, így a programot összesen kb. 810 sorra növelik, de a **hoc**-ot egy egyszerű számológépből egy programozási nyelvvé alakítják át. Nem fogunk most bemutatni minden sort. A 3. függelékben ugyanis megtalálhatjuk a teljes program listáját, így az egyes részek illeszkedését ott megnézhetjük.

A nyelvtanban a függvényhívások felelnek meg kifejezéseknek, az eljárás hívások pedig utasításoknak. A 2. függelékben mindegyiket részletesen elmagyarázzuk, és mutatunk néhány további példát is. Példaként itt egy, az argumentumában adott számnál kisebb Fibonacci-számokat kiíró eljárás definícióját és használatát adjuk meg:

```
$ cat fib
proc fib() {
        a = 0
        b = 1
        while (b < $1) {
                print b
                c = b
                b = a+b
                a = c
        }
        print "\n"
}
$ hoc6 fib -
fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
...
```

Ez a példa egyben a fájlok használatát is mutatja: a mínuszjel (–) fájlnév helyén a szabványos bemenetet jelenti.

Íme a faktoriális függvény:

```
$ cat fac
func fac() {
        if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
$ hoc6 fac -
```

```
fac(0)
        1
fac(7)
        5040
fac(10)
        3628800
...
```

A függvényeken vagy az eljárásokon belül az argumentumokra **$1** stb. módon hivatkozhatunk, akárcsak a burokban, de itt értékeket is adhatunk nekik. A függvények és eljárások rekurzívak, de csak az argumentumok lokális változók; az összes többi változó globális, azaz a program bármely részéből elérhető.

A **hoc** megkülönbözteti a függvényt az eljárástól, mivel így egy ellenőrzési lehetőséghez jutunk. Ez jól jön a verem megvalósításakor, ugyanis nagyon könnyen megfeledkezhetünk egy return utasításról vagy egy felesleges kifejezést felejthetünk a veremben.

A **hoc5** nyelvtanából meglehetősen sok változtatással állítottuk elő a **hoc6** nyelvtanát, de ezeket meghatározott helyekre csoportosítottuk. Használunk új tokeneket és nemterminális szimbólumokat ezenkívül a **%union** deklarációban egy új elemet, az argumentum számláló helyét:

```
$ cat hoc.y
...
%union {
        Symbol  *sym;       /* szimbólumtáblázat-mutató */
        Inst    *inst;      /* gépi utasítás */
        int     narg;       /* az argumentumok száma */
}
%token  <sym>   NUMBER STRING PRINT VAR BLTIN UNDER WHILE IF ELSE
%token  <sym>   FUNCTION PROCEDURE RETURN FUNC PROC READ
%token  <narg>  ARG
%type   <inst>  expr stmt asgn prlist stmtlist
%type   <inst>  cond while if begin end
%type   <sym>   procname
%type   <narg>  arglist
...
list:     /* üres */
        | list '\n'
        | list defn '\n'
        | list asgn '\n'    { code2(pop, STOP); return 1; }
        | list stmt '\n'    { code(STOP); return 1; }
        | list expr '\n'    { code2(print, STOP); return 1; }
        | list error '\n'   { yyerrok; }
        ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
        | ARG '=' expr
            { defnonly("$"); code2(argassign,(Inst)$1); $$=$3; }
        ;
```

```
stmt:       expr { code(pop); }
          | RETURN { defnonly("return"); code(procret); }
          | RETURN expr
                { defnonly("return"); $$=$2; code(funcret); }
          | PROCEDURE begin '(' arglist ')'
              { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
          | PRINT prlist {$$ = $2; }
...
expr:       NUMBER  { $$ = code2(constpush, (Inst)$1); }
          | VAR     { $$ = code3(varpush, (Inst)$1, eval); }
          | ARG     } defnonly("$"); $$ = code2(arg, (Inst)$1); }
          | asgn
          | FUNCTION begin '(' arglist ')'
              { $$ = $2; code3(call,(Inst)$1,(Inst)$4; }
          | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
...
begin:      /* üres */      { $$ = progp; }
          ;
prlist:     expr              { code(prexpr); }
          | STRING            { $$ = code2(prstr, (Inst)$1; }
          | prlist ',' expr   { code(prexpr); }
          | prlist ',' STRING { code2(prstr; (Inst)$3); }
          ;
defn:       FUNC procname { $2->type=FUNCTION; indef=1; }
            '(' ')' stmt  { code(procret); define($2); indef=0; }
          | PROC procname { $2->type=PROCEDURE; indef=1; }
            '(' ')' stmt    code(procret); define($2); indef=0; }
          ;
procname:   VAR
          | FUNCTION
          | PROCEDURE
          ;
arglist:    /* üres */      { $$ = 0; }
          | expr            { $$ = 1; }
          | arglist ',' expr { $$ = $1 + 1; }
          ;
%%
...
```

Az **arglist**-nél alkalmazott szabály az argumentumok számát állapítja meg. Első látásra úgy tűnhet, hogy az argumentumokat valahogy össze kell gyűjteni, de ez nem így van, hiszen az argumentumlistában szereplő minden egyes kifejezés pontosan ott helyezi el az értékét a veremben, ahol arra szükség van. Csak azt kell tudnunk, hogy hány argumentum van a veremben.

A **defn** szabályai a **yacc** egy új lehetőségét, a beágyazott teendőket vezetik be. Ezzel egy teendőt egy szabály közepébe helyezhetünk, így azt a **yacc** program a szabály felismerése alatt fogja végrehajtani. Ezt a lehetőséget használjuk ki annak

megjegyzésére, hogy egy függvény vagy egy eljárás definíciójában vagyunk éppen. (Egy másik lehetőség volna, ha a **begin** analógiájára létrehoznánk egy új szimbólumot, amit kellő időben fel lehet ismerni.) A **defnonly** függvény figyelmeztető üzenetet ír ki, ha függvényen vagy eljáráson kívül olyan szerkezet fordul elő, amely ott nem szerepelhet. Gyakran választhatunk, hogy szintaktikusan vagy szemantikusan csípjük-e el a hibákat (a definiálatlan változók kezelésénél már találkoztunk ezzel a kérdéssel). A **defnonly** függvény jó példa arra, hogy a szemantikai ellenőrzés kézenfekvőbb lehet, mint a szintaktikai.

```
defnonly(s)     /* figyelmeztet a nemmegengedett definícióra */
        char *s;
{
        if (!indef)
          execerror(s, "used outside definiton");
}
```

Az **indef** változót a **hoc.y**-ban vezettük be és a **defn**-beli tevékenységek állítják be.

A lexikai elemző kibővült az argumentumok ellenőrzésével – a $ jelet egy szám követi-e –, és az idézőjeles szövegek ellenőrzésével. A szövegekben előforduló fordított törtvonalas láncokat – pl. \n-t – a **backslash** függvény értelmezi.

```
yylex()     /* hoc6 */
...
      if (c == '$') { /* argumentum? */
            int n = 0;
            while (isdigit(c=getc(fin)))
                n = 10 * n + c - '0';
            ungetc(c, fin);
            if (n == 0)
              execerror("strange $...", (char *)0);
            yylval.narg = n;
            return ARG;
      }
      if (c == '"') {        /* idézőjeles szöveg */
            char sbuf[100], *p, *emalloc();
            for (p = sbuf; (c=getc(fin)) != '"'; p++) {
                if (c == '\n' || c == EOF)
                      execerror("missing quote", "");
                if (p >= sbuf + sizeof(sbuf) - 1) {
                      *p = '\0';
                      execerror("string too long", sbuf);
                }
                *p = backslash(c);
            }
            *p = 0;
            yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
            strcpy(yylval.sym, sbuf);
            return STRING;
      }
...
```

```
backslash(c)  /* a következő \'-es karakter értelmezése */
        int c;

{
        char *index(); /* bizonyos rendszerekben: `strchr()' */
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if (c != '\\')
                return c;
        c = getc(fin);
        if (islower(c) && index(transtab, c))
                return index(transtab, c)[1];
        return c;
}
```

A lexikai elemző egy *véges állapotú* automatára példa, akár C nyelvben íródott, akár egy **lex**-hez hasonló programgenerátor állította elő. A mi *ad hoc* C nyelvű változatunk már elég bonyolult lett; továbbhaladva a **lex** valószínűleg jobb mind a forráskód sorainak száma, mind a változtatás egyszerűsége tekintetében.

A többi módosítás legnagyobb része a **code.c**-ben van, ezenkívül a **hoc.h**-t bővítettük néhány függvénynévvel. A gép azonos az előzővel, de az egymásba ágyazott függvény- és eljáráshívások nyomkövetéséhez bevezettünk egy második vermet. (Egy második verem kezelése egyszerűbb, mint a már létezőt tovább zsúfolni.) Következzen itt a **code.c** első része:

```
$ cat code.c
#define     NPROG 2000
Inst        prog[NPROG];        /* a gép */
Inst        *progp;             /* a következő szabad lyuk a kód generálásához */
Inst        *pc;                /* a programszámláló a végrehajtás idején */
Inst        *progbase = prog;   /* az aktuális alprogram kezdete */
int         returning;          /* 1, ha return utasítás volt */
typedef     struct Frame  {     /* eljárás/függvény hívások verem kerete */
            Symbol  *sp;        /* a szimbólumtáblázat eleme */
            Inst    *retpc;     /* ahova return után visszatér */
            Datum   *argn;      /* a verem n-edik argumentuma */
            int     nargs;      /* az argumentumok száma */
} Frame;
#define     NFRAME 100
Frame       frame[NFRAME];
Frame       *fp;                /* a keret mutatója */
initcode() {
            progp = progbase;
            stackp = stack;
            fp = frame;
            returning = 0;
}
...
$
```

Mivel a szimbólumtáblázat most már tartalmazza az eljárások és a függvények, valamint a kiírandó szövegek mutatóit, ezért a **hoc.h**-beli **union** típust kibővítjük:

```
$ cat hoc.h
typedef struct Symbol          { /* a szimbólumtáblázat eleme */
        char *name;
        short type;
        union {
                double   val;          /* VAR */
                double   (*ptr)();     /* BLTIN */
                int      (*defn)();    /* FUNCTION, PROCEDURE */
                char      *str;        /* STRING */
        } u;
        struct Symbol *next;   /* csatlakozás egy másikhoz */
} Symbol;
...
$
```

A fordítás során a szimbólumtáblázatba a **define** segítségével beírunk egy függvényt, amely sikeres fordítás esetén tárolja az előállított kód kezdőpontját a táblázatban, és módosítja a kód utáni első szabadhely-mutatót.

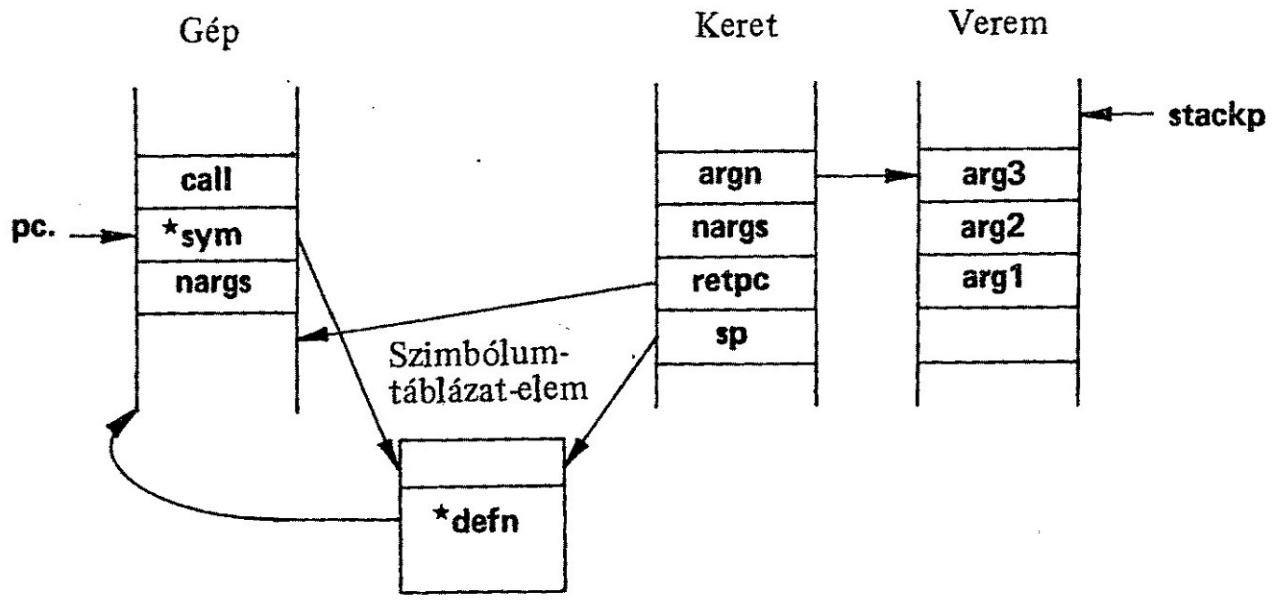
```
define(sp)     /* a func/proc elhelyezése a szimbólumtáblázatba */
        Symbol *sp;
{
        sp->u.defn = (Inst)progbase;   /* a kód kezdete */
        progbase = progp;   /* itt kezdődik a következő kód */
}
```

Amikor a végrehajtás során egy függvényt vagy eljárást meghívunk, akkor már minden argumentumnak kiszámítva a veremben kell lennie (az első argumentum van legalul). A hívás opcode-ját a szimbólumtáblázat-mutató és az argumentumok száma követi. A **Frame** a verembe kerül, ez minden lényeges információt tartalmaz a rutinról – hívás után a szimbólumtáblázat melyik elemére kell visszatérni, a veremben hol találhatók az argumentumok, és a híváskor hány argumentumot adtunk meg. A keretet a **call** állítja elő, amely azután végrehajtja a rutin kódját.

```
call()              /* call függvény */
{
    Symbol *sp = (Symbol *)pc[0];   /* a függvény szimbólumtáblázatának eleme */
    if (fp++ >= &frame[NFRAME-1])
      execerror(sp->name, "call nested too deeply");
    fp->sp = sp;
    fp->nargs = (int)pc[1];
    fp->retpc = pc + 2;
    fp->argn = stackp - 1;   /* az utolsó argumentum */
    execute(sp->u.defn);
    returning = 0;
}
```

Ezt a szerkezetet mutatjuk be a 8.2. ábrán.

Végül a **procret** vagy **funcret** végrehajtásával visszatér a meghívott eljárás:



8.2. ábra. Az eljáráshívás adatszerkezetei

```
funcret()       /* függvényből való visszatérés */
{
        Datum d;
        if (fp->sp->type == PROCEDURE)
          execerror(fp->sp->name, "(proc) returns value");
        d = pop();  /* megőrzi a függvény visszaadott értékét */
        ret();
        push(d);
}

procret()       /* eljárásból való visszatérés */
{
        if (fp->sp->type == FUNCTION)
       execerror(fp-> sp-> name,
            "(func) returns no value");
        ret();
}
```

A ret függvény kiveszi az argumentumokat a veremből, tárolja az fp keretmutatót és beállítja a programszámlálót.

```
ret()       /* függvény és eljárás közös visszatérése */
{
    int i;
    for (i = 0; i < fp->nargs; i++)
        pop();  /* argumentumok kivétele */
    pc = (Inst *)fp->retpc;
    --fp;
    returning = 1;
}
```

Az értelmező több rutinját egy kicsit ki kell egészíteni ahhoz, hogy az egymásba ágyazott utasításban előforduló **return**-t is tudják kezelni. Ezt a **returning** jelzővel nem elegánsan ugyan, de elfogadhatóan megoldottuk. A jelző akkor jelez igazat, ha egy **return** utasítást látott. Ekkor az **ifcode**, **whilecode** és **execute** korábban ér véget; a **call** a **returning** jelzőt nullázza.

```
ifcode()
{
        Datum d;
        Inst *savepc = pc;      /* then rész */
        execute(savepc+3);      /* feltétel */
        d = pop();
        if (d. val)
          execute(*((Inst **)(savepc)));
        else if (*((Inst **)(savepc+1)))   /* else rész? */
          execute(*((Inst **)(savepc+1)));
        if (!returning)
          pc = *((Inst **)(savepc+2)); /* következő utasítás */
}
whilecode()
{
        Datum d;
        Inst *savepc = pc;
        execute(savepc+2);   /* feltétel */
        d = pop();
        while (d.val) {
          execute(*((Inst **)(savepc))); /* mag */
          if (returning)
              break;
          execute(savepc+2);   /* feltétel */
          d = pop();
        }
        if (!returning)
          pc = *((Inst **)(savepc+1)); /* következő utasítás */
}
execute(p)
        Inst *p;
{
        for (pc = p; *pc != STOP && !returning; )
          (*(*pc++))();
}
```

Az argumentumokat értékadáshoz vagy felhasználáshoz a **getarg** veszi elő a veremből, ez pontosan kezeli a vermet:

```
double *getarg()      /* visszatérési mutató az argumentumhoz */
{
        int nargs = (int) *pc++;
        if (nargs > fp->nargs)
          execerror(fp->sp->name, "not enough arguments");
```

```
        return &fp->argn[nargs - fp->nargs].val;
}
arg()       /* argumentum elhelyezése a verembe */
{
        Datum d;
        d.val = *getarg();
        push(d);
}
argassign()     /* a verem tetejének tárolása az argumentumban */
{
        Datum d;
        d = pop();
        push(d);  /* az érték a veremben marad */
        *getarg() = d.val;
}
```

Számok és szövegek kiírását a **prexpr** és a **prstr** végzi:

```
prstr()       /* szöveg kiírása */
{
        printf("%s", (char *) *pc++);
}

prexpr()      /* számérték kiírása */
{
        Datum d;
        d = pop();
        printf("%.8g ", d. val);
}
```

A **varread** függvény a változókat olvassa be; 0-val tér vissza, ha fájl végét talált, egyébként 1-gyel, és értéket ad a megadott változónak.

```
varread()     /* a változóba olvas */
{
        Datum d;
        extern FILE *fin;
        Symbol *var = (Symbol *) *pc++;
Again:
        switch (fscanf(fin, "%lf", &var->u.val)) {
        case EOF:
                if (moreinput())
                  goto Again;
                d.val = var->u.val = 0.0;
                break;
        case 0:
                execerror("non-number read into", var->name);
                break;
```

```
        default:
                d.val = 1.0;
                break;
        }
        var->type = VAR;
        push(d);
}
```

Ha az aktuális bemeneti fájl véget ér, akkor a **varread** meghívja a **moreinput**-ot, amelyik megnyitja a következő argumentumfájlt, ha van olyan. A **moreinput** tanulmányozásával az itt elmondottaknál többet tudhatunk meg a bemenet feldolgozásáról; a részletek a 3. függelékben találhatók meg.

Ezzel a **hoc** fejlesztésének a végéhez jutottunk! Összehasonlításként felsoroljuk az egyes változatok nemüres sorainak számát:

```
hoc1   59
hoc2   94
hoc3  248      (a lex-változat 229)
hoc4  396
hoc5  574
hoc6  809
```

Természetesen programok számították ki a sorszámokat:

```
$ sed '/^$/d' `pick *.[chyl]` | wc -l
```

A nyelvet semmiképp nem tekintjük befejezettnek, hiszen könnyen kitalálhatnánk még hasznos továbbfejlesztéseket, azonban nem ez a célunk. A következő feladatokkal csupán felhívjuk a figyelmet néhány, hasznosnak ígérkező továbbfejlesztésre.

8-18. feladat. Változtassuk meg a **hoc6**-ot úgy, hogy a szubrutinoknál fogadja el a megnevezett formális paramétereket a **$1** stb. alakok alternatívájaként! □

8-19. feladat. Mint tudjuk, a paraméterek kivételével az összes változó globális. A verem lokális változókkal való kibővítésének mechanizmusa nagyjából már kész van. Egy lehetséges megközelítés: bevezetünk egy **auto** deklarációt, amelyik a veremben helyet foglal a felsorolt változók részére; és a fel nem sorolt változókat globálisnak tekintjük. A szimbólumtáblázatot is ki kell bővíteni, hogy először a lokális változók között keressünk, és csak utána a globálisak között. Hogyan kapcsolódik ez a megnevezett argumentumokhoz? □

8-20. feladat. Hogyan lehetne a **hoc**-ot tömbökkel bővíteni? Hogyan lehetne ezeket a tömböket a függvényeknek és eljárásoknak átadni? Hogyan lehetne visszaadni ezeket? □

8-21. feladat. Általánosítsuk a karakterlánc-kezelést úgy, hogy a változókban számok helyett szövegek is lehessenek! Milyen operátorokra lenne szükség? A feladat nehe-

zebb része a tárkezelés: biztosítanunk kell a karakterláncok olyan tárolását, amellyel a feleslegessé válók helye azonnal felszabadul, így a tárat nem pazaroljuk el. Átmeneti lépésként készítsünk jobb kimenetformázó lehetőséget! Pl. tegyük elérhetővé a C nyelv **printf** utasításának egyes alakjait! □

8.7. A hatékonyság vizsgálata

A **hoc** programunkat értékelve összehasonlítottuk a futását más UNIX számolóprogramok futásával. A következő táblázatot ugyan fenntartással kell kezelni, de jelzi programunk ésszerűségét. Minden idő másodpercekben megadott PDP–11/70-es felhasználási időt jelent. Az összehasonlítást két feladaton végeztük. Az első az **ack(3,3)** Ackermann-függvény kiszámítása. Ez a függvényhívási mechanizmust teszi próbára; 2432 hívást hajt végre, néhányszor elég mélyen egymásba ágyazva.

```
funck ack() {
        if ($1 == 0) return $2+1
        if ($2 == 0) return ack($1-1, 1)
        return ack($1-1, ack($1, $2-1))
}
ack(3,3)
```

A második feladatban százszor kiszámítjuk az 1000-nél kisebb Fibonacci-számokat; ez leginkább aritmetikai műveleteket jelent, egy-két függvényhívással.

```
proc fib() {
        a = 0
        b = 1
        while (b < $1) {
                c = b
                b = a+b
                a = c
        }
}
i = 1
while (i < 100) {
        fib(1000)
        i = i+1
}
```

Az összehasonlításban szereplő négy nyelv: a **hoc**, a **bc(1)**, a **bas** (egy csak a PDP–11-en futó ősrégi BASIC dialektus) és a C nyelv (minden változót **double** típusúként használva).

A 8.1. táblázat értékei az összes felhasználói és rendszer CPU időt összegzik, ahogyan azt a **time** függvény mérte. Egy C nyelvű programot annak meghatározására is utasíthatunk, hogy az adott időből melyik függvény mennyit használt el. A progra-

8.1. táblázat

Felhasználói idők másodpercben
(PDP-11/70-en)

Program	**ack(3,3)**	**fib**(1000) százszor
hoc	5,5	5,0
bas	1,3	0,7
bc	39,7	14,9
C	<0,1	<0,1

mot „profilírozva" kell használni, azaz a C fordításhoz és minden betöltéshez meg kell adni a **−p** opciót. Ha úgy módosítjuk a **makefile**-t, hogy ezt olvassa:

```
hoc6: $(OBJS)
        cc $(CFLAGS) $(OBJS) −lm −o hoc6
```

akkor a **cc** parancs a **CFLAGS** változót használja. Ha ezután kiadjuk a

```
$ make clean; make CFLAGS=−p
```

parancsot, akkor így előáll a „profilírozott" program. Futásakor ez egy **mon.out** nevű fájlba gyűjti az adatokat, amelyeket a **prof** program értékel ki.

Az elmondottak bemutatására tegyünk próbát a **hoc6**-tal és a fenti Fibonacci-programmal:

```
$ hoc6 <fibtest                                          Tesztfuttatás.
$ prof hoc6 / sed 15q                                    Elemzés.
        name    %time     cumsecs      #call    ms/call
        _pop     15.6        0.85      32182       0.03
       _push     14.3        1.63      32182       0.02
      mcount     11.3        2.25
         csv     10.1        2.80
        cret      8.8        3.28
     _assign      8.2        3.73       5050       0.09
       _eval      8.2        4.18       8218       0.05
    _execute      6.0        4.51       3567       0.09
    _varpush      5.9        4.83      13268       0.02
         _lt      2.7        4.98       1783       0.08
    _constpu      2.0        5.09        497       0.22
```

```
_add        1.7          5.18          1683        0.05
_getarg     1.5          5.26          1683        0.05
_yyparse    0.6          5.30             3       11.11
$
```

A „profilírozással" kapott adatok éppúgy véletlen ingadozásokat mutatnak, mint a **time** függvényből származóak, tehát ezek is csak tájékoztató adatokként kezelendők, nem pedig abszolút igazságként. Az itt szereplő számok sugallják, hogyan lehetne a **hoc**-ot felgyorsítani, *ha erre szükség volna.* A futási idő egyharmada megy el a veremkezeléssel. Ha beszámítjuk a C szubrutinokat összekapcsoló **csv** és **cret** függvényekre eső időt is, akkor a „rezsi" még nagyobb lesz. (Az **mcount** rutint csak a **cc −p** kiegészítéssel fordított „profilírozás" használja.) Ha a függvényhívásokat makrókkal helyettesítenénk, akkor érezhető különbséget kapnánk.

E feltételezés kipróbálása érdekében átírtuk a **code.c**-t, veremkezelő makrókra cserélve a **push** és a **pop** hívásokat:

```
#define push(d) *stackp++ = (d)
#define popm() *--stackp  /* függvényre most is szükség van */
```

(A **pop** függvényre mint a gép egyik opcode-jára továbbra is szükség van, így nem cserélhetjük le mindegyik **pop**-ot.) Az új változat kb. 35 százalékkal gyorsabb; a 8.1. táblázatbeli idő 5,5 másodpercről 3,7-re, 5,0-ról pedig 3,1-re csökkent.

8-22. feladat. A **push** és **popm** makrók nem végeznek hibaellenőrzést. Okos dolog ez? Hogyan tudnánk egyesíteni a függvények nyújtotta hibakezelést a makrók gyorsaságával? □

8.8. Visszatekintés

Sok fontos dolgot megtanulhattunk ebből a fejezetből. Az első, hogy számunkra a nyelvfejlesztési eszközök áldást jelentenek. A könnyű kísérletezési lehetőség kihasználásával a munka érdekesebb részére, a nyelv tervezésére összpontosíthatunk. A nyelvtani leírás a megvalósítást is rendszerbe foglalja; a rutinokat a nyelvtan kapcsolja össze, és az eszközök a rutinokat az elemzés előrehaladtával mindig a megfelelő pillanatban hívják meg.

A második, inkább filozofikus tanulság az, hogy az adott feladatot érdemesebb nyelvfejlesztésnek tekinteni, mint „programozásnak". Ha egy programot úgy szervezünk meg, mint egy nyelv fordítóját, ez elősegíti, hogy pontos legyen a szintaxisa (azaz a felhasználói interfésze) és megvalósításának a szerkezete. Ez a felfogás elősegíti azt is, hogy könnyű legyen majd új sajátosságokat hozzátenni a már létezőkhöz. „Nyelvek" alatt persze nem csak hagyományos programnyelveket értünk; a mi gyakorlatunkban ilyenek az **eqn**, **pic**, **yacc**, **lex** és maga a **make** is.

Sokat megtudtunk az eszközök használatáról is, a **make** pl. felbecsülhetetlen értékű. Teljesen kiküszöböli a hibáknak azt a csoportját, amely a módosított programok újrafordításának elfelejtéséből származik. Elősegíti, hogy ne végezzünk felesleges munkát. Támogatja, hogy egyetlen fájlba szedjük össze az egymáshoz kapcsolódó és esetleg egymástól is függő műveleteket.

A fejlécfájlok jól kezelhetővé teszik azokat az adatdeklarációkat, amelyeket több fájlban is látni kell. Az információ központosításával kiszűrik az össze nem illő változatok okozta hibákat, különösen ha a **make**-kel együtt használjuk ezeket. Az is lényeges, hogy az adatok és a programok úgy legyenek fájlokba szervezve, hogy csak akkor legyenek láthatóak, amikor kell.

Sok minden van, amit hely hiányában nem emeltünk ki. Ilyen pl., hogy a **hoc** programcsalád kifejlesztése során milyen gyakran használtuk a többi UNIX eszközt. A program minden egyes változata külön jegyzék alá került, de az azonos fájlokat összeláncoltuk; az **ls**-t és a **du**-t gyakran használtuk arra, hogy nyomon kövessük, mi hol van. Sok más kérdést is program segítségével válaszoltunk meg. Emlékeztetésül felsorolunk néhányat.

Hol deklaráltunk egy adott változót? Segít a **grep**. Mit is módosítottunk ebben a változatban? Segít a **diff**. Hogy építettük be a módosításokat abba a változatba? Segít az **idiff**. Milyen nagy ez a fájl? Segít a **wc**. Itt az ideje, hogy készítsünk egy mentést? Segít a **cp**. Hogy tudnánk csak az utolsó mentés után készült változatokat elmenteni? Segít a **make**. Ez a hozzáállás jellemző a UNIX rendszerrel való mindennapos programfejlesztésre: egy sereg kisméretű eszközt használunk, egyenként vagy szükség esetén egymást kiegészítve. Ezek segítségével a gép végzi el azt a munkát, amit különben kézi úton kellene.

Történeti és irodalmi megjegyzések

A **yacc**-ot Steve Johnson fejlesztette ki. A nyelvek azon osztályát, amihez a **yacc** képes elemzőt generálni, **LALR(1)**-nek nevezik: balról jobbra elemző, a bemenetben legalább egy tokennel előre néz. A **yacc**-kal együtt született meg az az elgondolás, hogy külön leírás készüljön a precedenciára és a nyelvtani kétértelműségek feloldására. Lásd A. V. Aho, S. C. Johnson és J. D. Ullman: „Deterministic Parsing of ambiguous grammars" című cikkét (*CACM*, 1975., augusztus). Az elemzési táblázatok létrehozásához és kitöltéséhez néhány újszerű algoritmust, ill. adatstruktúrát is alkalmazunk.

A **yacc**-ot, ill. más, elemzőt előállító programot megalapozó elméletről jó leírás található A. V. Aho és J. D. Ullman könyvében: *Principles of Compiler Desing* (Addison-Wesley, 1977.) A **yacc**-ot magát a *UNIX kézikönyv 2B* kötete írja le. Ott szerepel egy, a **hoc2**-vel összemérhető számolóprogram is, tanulságos lehet a kettő egybevetése.

A **lex**-et eredetileg Mike Lesk írta. Az elméleti háttér szintén Aho és Ullman könyvében található, a **lex** nyelv leírása pedig a *UNIX kézikönyvben.*

A **yacc**-t, s kisebb részben a **lex**-et több fordítóprogram készítésénél használtuk,

ideértve a hordozható C fordítót, a Pascal, a FORTRAN 77, a Ratfor, az **awk**, a **bc**, az **eqn** és a **pic** fordítót.

A **make**-et Stu Feldman írta. L. a „MAKE – a program for maintaining computer programs" című cikket, a *Software – Practice & Experience* folyóirat 1979. áprilisi számában.

Jon Bentley könyve, a *Writing Efficient Programs* (Prentice-Hall, 1982) olyan módszereket ír le, amelyekkel felgyorsíthatók a programok. A hangsúly azon van, hogy először találjuk meg a megfelelő algoritmust, és ha kell, azután javítsunk a programon.

9. Dokumentumok készítése

A UNIX rendszer egyik első alkalmazása dokumentumok szerkesztése és formázása volt; a Bell Laboratóriumok vezetőségét valójában azzal győzték meg az első PDP-11-es gép vásárlásáról, hogy egy dokumentumkészítő rendszert hoznak rajta létre, nem pedig egy operációs rendszert. (Szerencsére többet kaptak, mint amit kialkudtak maguknak.)

Az első szövegformázó programot **roff**-nak nevezték. Ez egy kicsi, gyors és mindaddig könnyen kezelhető pogram volt, amíg valaki egyszerű dokumentumokat akart sornyomtatón készíteni. A következő szövegformázó program, a Joe Ossanna által készített **nroff** sokkal többet akart elérni. Ahelyett, hogy a felhasználók minden elképzelhető szövegformátumára igyekezett volna felkészülni, Ossanna az **nroff**-ot programozhatóra készítette. Így nagyon sok szövegformázó feladatot **nroff** nyelven írt programmal oldott meg.

Amikor 1973-ban egy kis betűszedő gépet szereztek be, az **nroff**-ot kibővítették, hogy a szedőgép többféle betűméretét és betűtípusát*, bővebb karakterkészletét is kezelje. Az új programot **troff**-nak hívták (amit az „en-roff" analógiájára, „tee-roff"-ként ejtenek ki). Az **nroff** és **troff** alapvetően azonos programok, és azonos bemeneti nyelvet használnak; az **nroff** egyszerűen figyelmen kívül hagyja azokat a parancsokat, amiket nem tud értelmezni (ilyen pl. a betűméret változtatására vonatkozó parancs). Elsősorban a **troff**-ról fogunk beszélni, de a legtöbb megjegyzésünk ugyanúgy vonatkozik az **nroff**-ra is, csupán az adott nyomtató korlátait kell figyelembe venni.

A **troff** legnagyobb erőssége az alapnyelv hajlékonysága és a programozhatóság – szinte bármilyen szövegformázási feladatra fel lehet készíteni. A rugalmasságnak nagy ára van – **troff**-ot gyakran megdöbbentően nehéz használni. Be kell vallani, hogy a legtöbb UNIX rendszer felügyelete alatt működő dokumentumkészítő programot arra tervezték, hogy a **troff**-ot kiegészítsék.

Egy példa erre a lap elrendezése. A dokumentum általános meghatározói: a címek, a fejlécek, a bekezdések formája, az oldalszámozás helye, egy lap mérete és így tovább. Ezeknek nincsen beépített értelmezése, ezeket be kell programozni. Nem kell minden egyes felhasználónak mindegyik dokumentumnál meghatároznia a részletkér-

* A szerzők a betűtípus szót a magyar szóhasználattól eltérően használják. A betűváltozatot (pl. *dőlt*, **félkövér** stb.) és a betűtípust (pl. Helvetica, Times stb.) egyaránt betűtípusnak nevezik. A UNIX rendszer megismerését ez nem zavarja, ezért meghagytuk az eredeti szóhasználatot. *(A szerk.)*

déseket. Egy szabványos formázó parancscsomag áll a rendelkezésére. A készlet használója nem azt mondja, hogy a „következő sor legyen középre igazítva és félkövér nagybetűkkel írva", hanem azt, hogy „a következő sor egy cím", s a **troff** csomagból használja majd fel a cím formai leírását. A felhasználó a méretek, típusok és helyzetek helyett a dokumentum logikai összetevőit – címek, fejlécek, bekezdések, lábjegyzetek stb. – adja meg.

Sajnálatos módon a formázóparancsok „szabványos" készletnek indult készlete ma már nem szabványos: többféle, széles körben használt csomag mellett sok helyi változat is létezik. A következőkben két általános célú csomagot mutatunk be: az **ms**-t, az eredeti „szabványt" és ennek egy újabb változatát, az **mm**-et, amely System V. szabványa. Bemutatjuk a **man** csomagot is, ami a UNIX kézikönyv szabványa szerinti programleíró lap írásához használható.

Az **ms**-re fogunk összpontosítani, ami a UNIX 7. kiadásában a szabványos, és az összes ilyen csomag használatára példát ad, valamint elég hatékony a feladatok megoldásához. (Ennek segítségével nyomtatták ki az eredeti könyvet is.) Bár egy kicsit ki kellett bővítenünk, pl. a különleges betűtípusú parancsneveket, utasításokat a szövegben előforduló kezelőparanccsal.

Ez a jellemző gyakorlat, a makrocsomagok legtöbb szerkesztési feladathoz jól használhatók, de néha vissza kell térni a makrókat megalapozó **troff** parancsok szintjére. Itt a **troff**-nak csak egy kis részét mutatjuk be.

Bár a **troff** lehetőséget ad a kimenet formájának tetszőleges kialakítására, nehézkesen használható pl. a matematikai képletek, a táblázatok és az ábrák esetében. E területek egyenként is annyi bonyodalmat jelentenek, mint egy dokumentum lapjának elrendezése. Ezeket a feladatokat viszont más formában kezeljük. Formázó parancscsomagok helyett különleges célnyelveket fejlesztettek ki a matematikai képletekhez, táblázatokhoz és ábrákhoz, amelyek segítségével könnyen leírható, amit akarunk. Ezek mindegyikét külön program kezeli, amely az adott nyelvet **troff** parancsokra fordítja le. A programok UNIX csöveken keresztül kapcsolódnak egymáshoz.

Ezen preprocesszorprogramok jól példázzák a UNIX feladatmegoldási elvét is: ahelyett, hogy a **troff**-ot még nagyobbá és a jelenleginél is bonyolultabbá tennénk, vele együttműködő, de különálló programokat írunk. (Kifejlesztésükhöz természetesen felhasználtuk a 8. fejezetben leírt nyelvfejlesztési eszközöket.) Két programot fogunk leírni: a **tbl**-t, amely táblázatokat alakít ki és az **eqn**-t, amely matematikai kifejezéseket kezel.

Megpróbálunk útmutatást adni a dokumentumok készítéséhez és az ezt támogató eszközökhöz. A fejezetben szereplő példákat a 8. fejezet **hoc** nyelvének leírása, valamint egy **hoc** kézikönyv szolgáltatja. E dokumentum kinyomtatva megtalálható a könyv második függelékében.

9.1. Az ms makrocsomag

A makrocsomagok használatának alapvető igénye az volt, hogy a dokumentumokat logikai részeik – cím, alcím, szakasz – megadásával írjuk le, nem pedig olyan részletekkel, mint a sortávolság, a betűk típusa és mérete. Ez megment bennünket egy

sor igen nehéz munkától, és leválasztja a dokumentumról a tárgy lényegét nem érintő részleteket; sőt, azonos logikai nevű, de eltérő tartalmú makrocsomag-definíciókkal elérhetjük, hogy a dokumentumunk teljesen másképp jelenjen meg. Pl. ugyanazon formázóparancsokkal, de négy különböző makrocsomaggal formázva a dokumentum végigmehet a következő fázisokon: kutatási jelentés, konferenciaelőadás, folyóiratcikk, könyvbeli fejezet.

A **troff** bemenete – akár használunk makrocsomagot, akár nem – közönséges szöveg, amelybe formázóparancsokat vegyítünk. A parancsoknak két típusa van. Az első a sor elején levő ponttal kezdődik, amit egy vagy két betű vagy számjegy és esetleg paraméterek követnek; ahogy ez a következőkben is látszik:

```
.PP
.ft B
Ez egy kisbetűs félkövér betűtípusú bekezdés.
```

A **troff** beépített parancsainak kisbetűs neve van, így a makrocsomag parancsainak egyezményesen nagybetűsek a nevei. Ebben a példában a **.PP** a bekezdés **ms**-beli parancsa, és a **.ft B** egy **troff** parancs, ami **félkövér** betűtípusra vált át. (A betűtípusoknak nagybetűs nevük van; az elérhető betűtípusok különböző betűszedőkön különbözők lehetnek.)

A **troff** parancsok másik alakja egy fordított törtvonallal (\) kezdődő karaktersorozat, amely a bemenet bármely részében előfordulhat. Pl. a **\fB** szintén a félkövér betűtípusra állít át. A parancsok ezen formája igazi **troff** alak; röviden erre még visszatérünk.

Egy dokumentum tagolásához már az is elegendő lehet, ha minden bekezdés előtt egy **.PP** parancsot adunk ki. A legtöbb dokumentum előállításához elegendő egy tucatnyi **ms** parancs. Pl. a **hoc** leírását tartalmazó 2. függelékben a következők találhatók: bekezdések, cím, a szerzők neve, a tartalmi kivonat, automatikusan beszámozott szakaszcímek. Csak 14 különböző parancsot használ, ezek közül több párban szerepel. A cikk általános alakja az **ms** parancsokkal a következő:

```
.TL
A dokumentum címe (egy vagy több sor).
.AU
A szerzők neve, soronként egy
.AB
Kivonat, a .AE jelzi a végét.
.AE
.NH
Sorszámozott címek (automatikus számozás).
.PP
Bekezdés...
.PP
Másik bekezdés...
.SH
Alcím (nem számozott).
.PP
...
```

A formameghatározó parancsnak a sor elején kell állnia. A parancsok közötti bemenet szabad formátumú: a bemenetben szereplő újsor-karakterek helye nem lényeges, mivel a sorok hosszának kialakítása során a **troff** a szavakat átviszi az egyik sorból a másikba (az eljárás neve *filling:* feltöltés), és a margóhoz való igazítás után a fennmaradó üres helyet egyenletesen elosztja a szavak között (*justification:* kiegyenlítés). Hasznos dolog minden mondatot új sorban kezdeni, ezáltal az újabb javításokat könnyebb elvégezni.

Következő példánk a jelenlegi **hoc** leírás kezdete:

```
.TL
Hoc – Interaktív lebegőpontos aritmetikai nyelv
.AU
Brian Kernighan
Rob Pike
.AB
A
.I hoc
lebegőpontos kifejezések egyszerű
programozható értelmezője.
C nyelvhez hasonló vezérlés-ellenőrzése,
függvénydefiníciója van és rendelkezik a
szokásos beépített numerikus
függvényekkel, mint pl. a koszinusz
és a logaritmus.
.AE
.NH
Kifejezések
.PP
A
.I hoc
a C nyelvhez hasonlóan egy kifejezésnyelv:
van ugyan számos vezérlés-ellenőrző utasítása,
de a legtöbb utasítása, (mint pl. az értékadások),
olyan kifejezés, amelynek értéke közömbös.
...
```

A **.I** parancs az argumentumát dőlt betűkkel írja ki, ill. ha nincs argumentuma, akkor dőltbetűs írásra vált.

Ha használunk makrocsomagot, akkor azt a **troff** hívásakor argumentumként meg kell adni:

```
$ troff -ms hoc.ms
```

A **-m** utáni karakterek határozzák meg a makrocsomagot.*

Az **ms**-sel formázott **hoc**-leírás így néz ki:

* Az **ms** makrók a **/usr/lib/tmac/tmac.c** fájlban, a **man** makrók a **/usr/lib/tmac/tmac.man** fájlban találhatók.

Hoc – Interaktív lebegőpontos aritmetikai nyelv

Brian Kernighan
Rob Pike

KIVONAT

A *hoc* lebegőpontos kifejezések egyszerű programozható értelmezője. C jellegű vezérlés-ellenőrzése, függvénydefiníciója van és rendelkezik a szokásos beépített numerikus függvényekkel, mint pl. a koszinusz és a logaritmus.

1. Kifejezések

A *hoc* a C-hez hasonlóan egy kifejezésnyelv: van ugyan számos vezérlés-ellenőrző utasítása, de a legtöbb utasítása, mint pl. az értékadások, olyan kifejezés, amelynek értéke közömbös.

Kivetítés

A **troff** egyenletes sorkitöltése és margóra igazítása általában kényelmes, néha azonban nincs erre szükségünk. Pl. a programokat nem jó a margóra igazítani. Az ilyen fomázatlan anyagokat kivetítendő (display) szövegeknek hívják. A **.DS** (display start) és **.DE** (display end) **ms** parancsok a közrezárt szöveget úgy írják ki, ahogyan vannak, kicsit beljebb, de átrendezés nélkül. Íme a **hoc** kézikönyv következő része, amely egy rövid kivetítést is tartalmaz:

```
.PP
A
.I hoc
a C nyelvhez hasonlóan egy kifejezésnyelv;
van ugyan számos vezérlés-ellenőrzés utasítása,
de a legtöbb utasítása (mint pl. az értékadások),
olyan kifejezés, amelynek értéke közömbös.
Például az = értékadó operátor a jobb oldali operandus
értékét hozzárendeli a bal oldali
operandushoz, tehát értéket közvetít.
A többszörös értékadás is működik.
A kifejezés nyelvtana a következő:
.DS
.I
kifejezés:      szám
        |  változó
        |  (kifejezés)
        |  kifejezés binop kifejezés
```

```
        / unop kifejezés
        / függvény (argumentumok)
.R
.DE
A számok lebegőpontosak.
```

Ez így íródik ki:

A hoc a C-hez hasonlóan egy kifejezésnyelv: van ugyan számos vezérlés-ellenőrző utasítása, de a legtöbb utasítása, mint pl. az értékadások, olyan kifejezés, amelynek az értéke közömbös. Például az = értékadó operátor a jobb oldali operandus értékét hozzárendeli a bal oldali operandushoz, és így egy értéket ad; a többszörös értékadás is működik. A kifejezés nyelvtana:

kifejezés: *szám*
/ változó
/ (kifejezés)
/ kifejezés binop kifejezés
/ unop kifejezés
/ függvény (argumentumok)

A számok lebegőpontosak.

A kivetítés belsejében levő szöveget a **troff** általában nem tölti fel és nem igazítja lapszélre. Továbbá, ha az éppen használt lapon nincs elég hely, akkor a kivetítés anyagát (és mindent, ami utána következik) a következő lapra ír. A **.DS**-nek különféle opciói lehetnek, többek között az **L**-t a balra igazítására, a **C**-t a sorok egyenkénti középre igazítására, és a **B**-t az egész kivetítés középre helyezésére.

A fenti kivetítés szövegrészeit tabulátorok választják el. Alapértelmezésként a **troff** tabulátorai fél hüvelykenként követik egymást, nem a szokásos nyolc betűnyi távolságra. Ezt a karakterek változó szélessége magyarázza, ugyanis ha a tabulálás minden nyolcadik betűhelyen lenne, a karakterek különböző szélessége miatt a troff nem mindig úgy jelenítené meg a tabulálást, ahogyan elvárnánk.

Betűtípusváltás

Az **ms** makrók közt háromféle betűtípusváltó parancs van. A **.R** a szokásos álló betűkre vált át, a **.I** a *dőlt* betűkre és a **.B** a **kövér** betűkre. Mindegyik parancs az őt követő szöveg betűtípusát adja meg:

```
Ezek álló betűk, de
.I
```

```
ezek dőlt betűk
.R
ezek újra álló betűk, és
.B
ezek kövér betűk.
```

A **troff** ezt így fogja kinyomtatni:

Ezek álló betűk, de *ezek dőlt betűk*, ezek újra álló betűk, és **ezek kövér betűk.**

A **.I**-nek és a **.B**-nek lehet argumentuma, ekkor a betűtípusváltás csak az argumentumra érvényes. A **troff**-ban a szóközt tartalmazó argumentumokat idézőjelbe (") kell tenni, de csak ez az egy különleges karakter használható.

```
Ezek álló betűk, de
.I ezek
dőlt betűk, és
.B "ezek a szavak"
kövér betűsek.
```

Ezt a **troff** így írja ki:

Ezek álló betűk, de *ezek* dőlt betűk, és **ezek a szavak** kövér betűsek.

Végül megjegyezzük, hogy a **.I** és a **.B** második argumentumát álló betűvel írja ki, közvetlenül az elsőt követve. Ezt a lehetőséget általában arra használjuk, hogy az írásjeleket a megfelelő betűtípussal írjuk. Hasonlítsuk össze a

```
(zárójelezett
.I "dőlt betűs szavak)"
```

alábbi, hibás kiíratását:

(zárójelezett *dőlt betűs szavak)*

az itt következő

```
(zárójelezett
.I "dőlt betűs szavak")
```

helyes változatával:

(zárójelezett *dőlt betűs szavak*)

Az **nroff** is felismeri a betűtípus-eltéréseket, de az eredmény már nem ilyen szép. A dőlt betűk helyett aláhúz, nincs kövér betűje, bár az **nroff** néhány változata újbóli átírással szimulálja a kövér betűket.

A lábjegyzeteket a **.FS** vezeti be és a **.FE** zárja. Nekünk kell előállítani valamilyen azonosítójelet, mint amilyen egy csillag (*) vagy egy ilyen †. Ez a lábjegyzet a következő módon készült:

```
azonosítójelet, mint amilyen egy csillag (*) vagy egy ilyen tőr †. \(dg
.FS
\(dg Mint ez itt.
.FE
Ez a lábjegyzet a következő módon...
```

Beljebb tolt bekezdéseket a **.PI** paranccsal hozhatunk létre, a margón szerepelhet szám vagy más jel. A következő szövegrészt akarjuk előállítani:

(1) Az első rövid bekezdés.
(2) A második bekezdést azért írtuk hosszabbra, hogy bemutathassuk: a második sor ugyanúgy beljebb kezdődik, mint az első.

Ehhez a következő bemenetet kell megadnunk:

```
.IP (1)
Az első rövid bekezdés
.IP (2)
A második bekezdést...
```

Egy **.PP** vagy egy **.LP** (balra igazított bekezdés) megszünteti a **.IP** hatását. A **.IP** argumentuma bármilyen szöveg lehet: ha szóközöket is tartalmaz, akkor idézőjelek (") közé kell tenni. A beljebb tolás mértékének meghatározásához második argumentum is megadható.

A **.KS** és **.KE** parancspár szövegrészek együtt-tartására szolgál: ha az ezekkel közrezárt szöveg nem fér el az éppen íródó oldalra, akkor a nyomtatás új oldalra tér át. Ha **.KF**-et használunk **.KS** helyett, akkor a közrezárt szöveget követő részekből a program előrevesz annyit, hogy a közrezárt rész a következő lap tetejére kerüljön. A szerzők a **.KF**-et használták az eredeti könyv összes táblázatánál.

Az **ms** legtöbb alapértelmezés szerinti értékét megváltoztathatjuk az **ms** által használt **troff** változók, a *számregiszterek* átállításával. Talán a legegyszerűbbek a szöveg méretét és a sorok távolságát vezérlő regiszterek. A normál szövegméret a „10 pont", ahol egy pont kb. 1/72 inch $\approx$ 0,35 mm – ezt a mértékegységet a nyomdaipartól vettük át. A sorok általában 12 pont távolságra vannak egymástól. Ezeknek az értékeknek a módosításához pl. 9-re, ill. 11-re (ahogy a képernyőnknél szokásos) a PS és a VS számregisztereket kell átállítani:

```
.nr PS 9
.nr VS 11
```

Más számregiszterek is vannak, köztük az **LL** a sorhosszúságot, **PI** a bekezdés beljebb tolásának mértékét, **PD** a bekezdések közötti elválasztást adja meg. Ezek átállításának hatása a következő **.PP** vagy **.LP** parancsig terjed.

Az mm makrocsomag

Az **mm** makrocsomag részleteire itt nem térünk ki, mivel ez szellemében és gyakran részleteiben is nagyon hasonló az **ms**-hez. Több vezérlési lehetőségünk van a paramétereken keresztül, mint az **ms**-ben, többet tud (pl. automatikusan számozott listákat), és jobbak a hibaüzenetei. A 9.2. táblázat a 9.1. táblázatbeli **ms** parancsoknak megfelelő **mm** parancsokat tartalmazza.

9.1. táblázat

Gyakran használt **ms** parancsok (l. még **ms(7)**)

.AB	kivonat kezdete; **.AE** jelzi a végét
.AU	szerző neve a következő sorban; többszörös **.AU** megengedett
.B	kövérbetűs rész kezdete, ill. kövérítendő argumentum
.DS *t*	kivetítés kezdete (sorkiegyenlítés nélkül); **.DE** jelzi a végét, *t* = L (balra igazítva), C (középre téve), B (tömbként középre)
.EQ *s*	az *s* egyenlet kezdete (**eqn** bemenete); **.EN** jelzi a végét
.FS	lábjegyzet kezdete; **.FE** jelzi a végét
.I	dőlt betűs szöveg kezdete, ill. dőlt betűvel nyomtatandó argumentum
.IP *s*	beljebb tolt bekezdés, *s* a szélére kerül
.KF	összetartozó szövegrész, ha szükséges, folytatódhat a következő lapon; **.KE** jelzi a végét
.KS	azonos lapra kerülő szövegrész; **.KE** jelzi a végét
.LP	új, balra igazított bekezdés
.NH *n*	*n*-szinten sorszámozott cím; a cím a következő **.PP**-ig vagy **.LP**-ig tart
.PP	új bekezdés
.R	visszatérés az álló betűtípushoz
.SH	alcím; a cím **.PP**-ig tart
.TL	cím következik; a következő **ms** parancsig tart
.TS	táblázat kezdete (**tbl** bemenete); **.TE** jelzi a végét

9-1. feladat. A **.AE** vagy **.DE** lezáróparancs elhagyása általában katasztrófát okoz. Írjunk egy **mscheck** programot az **ms** (avagy a kedvenc makrocsomagunk) bemenetében előforduló hibák felfedezésére! Útmutatás: **awk**. □

9.2. táblázat

Gyakran használt **mm** parancsok

.AS	kivonat kezdete; **.RS** jelzi a végét
.AU	szerző neve következik első argumentumként
.B	kövér betűs rész kezdete, ill. kövérítendő argumentum
.DF	összetartozó szövegrész, ha szükséges, folytatódhat a következő lapon:
.DS	kivetítés kezdete; **.DE** jelzi a végét
.EQ	egyenlet kezdete (**eqn** bemenete); **.EN** jelzi a végét
.FS	lábjegyzet kezdete; **.FE** jelzi a végét
.I	dőlt betűs szöveg kezdete, ill. dőlt betűvel nyomtatandó argumentum
.H *n* "..."	*n*-szinten sorszámozott "..." cím
.HU "..."	sorszámozatlan "..." cím
.P	bekezdés. Használjuk a **.nr Pt 1**-et a beljebb tolt bekezdéshez
.R	visszatérés az álló betűtípushoz
.TL	cím következik; a következő **mm** parancsig tart
.TS	táblázat kezdete (**tbl** bemenete); **.TE** jelzi a végét

9.2. A troff szintje

A gyakorlatban néha az **ms, mm** vagy más makrocsomagok nyújtotta lehetőségek alapjához, magának a **troff**-nak a lehetőségeihez kell fordulnunk. Ez ahhoz hasonlít, mint ha assemblyben programoznánk, úgyhogy óvatosan és csak végső esetben szabad ezt tennünk.

Három esetben kerülünk ilyen helyzetbe: nem szokványos karakterek előfordulásakor, sorközi betűméret- vagy betűtípusváltáskor, és néhány alapvető formázási funkció esetében.

Karakternevek

A nem szokványos karakterek – görög betűk, mint pl. a π, grafikus jelek, mint pl. a ○ és †, valamint a sorok és szóközök sokasága – könnyen, de nem egyetlen séma szerint érhetők el. Minden ilyen karakternek van egy neve, amelyik vagy *c* alakú, ahol *c* egyetlen karakter, vagy *(cd* alakú, ahol *cd* egy karakterpár.

A **troff** a beírt ASCII mínuszjelet a kimenetén mínuszjel (−) helyett kötőjel-

9.3. táblázat

A **troff** parancs néhány különleges karaktere

–	kötőjel
\(hy	kötőjel, az előzővel azonos
\–	mínusz előjel, az éppen használt betűtípusban
\(mi	mínusz előjel, matematikai betűtípusban
\(em	gondolatjel
\&	semmi, csak védi a soreleji . karaktert
blank	ki nem tágítható szóköz
\¦	ki nem tágítható fél szóköz
\e	a különleges karaktersorozatot bevezető jel, általában \
\(bu	• korong
(dg	† tőr
\(*a	α. \(*b = β, \(*c = ξ, \(*p = π stb.
\f*X*	betűtípusváltás az *X* betűtípusra; *X* = P az előző típus
\f(*XX*	betűtípusváltás az *XX* típusra
\s*n*	*n* pontméretű írásjelekre vált; visszatér az előzőhöz, ha *n* = 0
\s±*n*	relatív pontméretváltoztatás

ként (-) írja. Az igazi mínuszjelet így kell beírni: \–, míg a gondolatjelet (—) így: \(*em*, ami az „em dash" rövidítése.

A 9.3. táblázat tartalmazza a leggyakoribb különleges karaktereket; a **troff** kézikönyvben ennél sokkal több van (az egyes rendszerek, pl. az Olvasó saját rendszerének a listája ettől eltérő is lehet).

Van, amikor azt kell mondani a **troff**-nak, hogy ne értelmezzen egy különleges karaktert, pl. egy fordított törtvonalat vagy egy soreleji pontot. A két leggyakoribb értelmezést kikapcsoló karakter a *e* és a *&*. A *e* sorozat értelmezés nélküli fordított törtvonalat ír ki, így kaphatunk a kimeneten fordított törtvonalat. A *&* semmit sem ír ki, ez egy nulla hosszúságú szóköz. Legfontosabb alkalmazása, hogy védje a **troff**-ot a sor elején álló pont értelmezésétől. Mi nagyon sokszor használtuk ebben a fejezetben a *e*-et és a *&*-et. Pl. a fejezet elején levő **ms** vázlatot a következőképpen gépeltük be:

```
\&.TL
.I "A dokumentum címe"
\&.AU
.I "A szerzők neve"
\&.AB
\&...
...
```

Az utóbbi sorokat természetesen így gépeltük be:

```
\e&.TL
\&.I "A dokumentum címe"
\e&.AU
...
```

és el lehet képzelni, hogy ezt hogyan kellett begépelni.

Egy másik – alkalmanként előforduló – különleges karakter, a ki nem tágítható szóköz, amely egy \-t követő szóköz. A **troff**, mivel a lap széléhez igazít, általában megnyújtja a szóközt, de a ki nem tágítható szóközt soha sem igazítja: ez olyan, mint a többi karakter, szélessége állandó. Arra is használható, hogy több szót egyetlen argumentumként adjunk meg:

```
.I A\ dokumentum\ címe
```

A betűtípus és a betűméret változtatása

A legtöbb betűtípus és betűméret változtatása a sor elején álló makrókkal – mint a **.I** – elvégezhető, de néha soron belül kell változtatni. Különösen gyakori ez az eset, ha az újsor-karakter a szóelválasztó karakter és a szó közepén kell betűtípust váltani. Ilyenkor a makrók használhatatlanok. Ebben a szakaszban megmutatjuk, hogy a **troff** ezt a feladatot hogyan oldja meg – megjegyezzük, hogy ezt a támogatást a **troff** nyújtja, nem pedig az **ms** makrocsomag.

A **troff** a fordított törtvonal karaktert használja a soron belüli parancsok bevezetésére. A két leggyakrabban használt parancs a betűtípusváltó **\f** és a betűméretet változtató **\s**.

Azt a betűtípust, amit a **\f**-fel kiválasztunk, az **f**-et közvetlenül követő karakter határozza meg. A

```
já\ fBtéko\flsan \fR\flössz\fBeválo\fRgatott \flbetűtípusok\fP
```

sor a következőképpen jelenik meg:

já**téko***san össz***eválo**gatott *betűtípusok*

A **\fP** váltással visszatérünk az előző betűtípusra – akármilyen is volt az utolsó váltás előtti típus. (Csak egy előző betűtípus van, nincs verem.)

Néhány betűtípusnak kétkarakteres neve van. Ezeket **\f(XX** formával adjuk meg, ahol az **XX** a betűtípus neve. Ha pl. egy betűszedőn a programsorok betűtípusát **CW**-nek hívják (ilyen volt a szerzők által használt!), akkor a kulcsszavakat a következőképpen kell írni:

```
\f(CWkulcsszó\fp
```

Folyton így begépelni elég fáradságos, ezért bővítsük ki az **ms**-t egy **.CW** makróval! Ezt a makrót használhatjuk a soron belüli kulcsszavak – mint pl. **troff** – kiszedéséhez:

```
A
.CW troff
formázó...
```

Azokat a formátumra vonatkozó döntéseket, amelyeket makrókkal írtunk le, később könnyen meg lehet változtatni.

A méretváltoztatást a **sn** sorozat vezeti be, ahol **n** az új méretet megadó egy vagy két számjegy: **s8** hatására 8-pontos méretre vált. Gyakran hasznosabb a relatív méretmeghatározás. Ezt a változtatást a méretszám elé tett mínusz vagy plusz előjellel adjuk meg. Pl. a KISBETŰ szót a következő módon lehet kiírni:

```
\s-2KISBETŰ\s0
```

A **s0** hatására a méret visszaáll a változtatás előtti értékre. Ez a·**fP** ikertestvére, de a **troff** hagyományai szerint nem **sP**-nek írják. A szerzők **ms** kiegészítésében volt erre egy **.UC** (upper case : nyomtatott nagybetü) makró.

A troff alapvető parancsai

A gyakorlatban még egy jó makrocsomag birtokában is egy sereg **troff** parancsot kell ismernünk, pl. a sortávolság és a feltöltés vezérlésére, a tabulálás beállítására és hasonlókra.

A **.br** parancs megszakítást okoz, azaz a **.br** utáni szövegrész a kimeneten egy új sorban fog megjelenni. Ezt pl. egy hosszú cím megfelelő helyen való szétválasztására használhatjuk:

```
.TL
Hoc – Interaktív lebegőpontos
.br
Aritmetikai nyelv
...
```

A **.nf** parancs felfüggeszti a kimenő sorok szokásos feltöltését, a bemenet minden sorát közvetlenül viszi át egy kimenő sorba. A **.fi** parancs visszaállítja a sorfeltöltést. A **.ce** parancs a következő sort középre helyezi.

A **.bp** parancs új lapot kezd. A **.sp** parancs egy üres sort visz a kimenetre. A **.sp** parancsot követheti argumentum, amely az üres sorok számát vagy az üresen hagyandó rész méretét adja meg.

.sp 3	3 sort hagy üresen.
.sp .5	Egy fél sort hagy üresen.
.sp 1.5i	1,5 hüvelyket hagy üresen.
.sp 3p	3 pontot hagy ki.
.sp 3.1c	3,1 centimétert hagy üresen.

A lap tetején üres helyet a **.sp** nem hagy, így a következő lapot elérő argumentum ugyanolyan hatású, mint egy **.bp**.

A **.ta** parancs a tabulálást állítja át (alapértelmezésben fél hüvelykenként követik egymást). A

.ta n n n...

a bal margótól számítva az adott távolságokra tabulálási helyeket állít be, akárcsak az **.sp**-nél minden megadott szám, ha **i** követi, hüvelyket (angolul inchet) jelent. Ha egy tabulálást megadó helyet **R** követi, akkor ez a szöveget a következő tabulálási helyhez jobbra igazítja; a **C** középre helyezett tabulálást jelent.

A **.ps n** parancs a pontméretet **n**-re állítja; a **.ft X** parancs a betűtípust **X**-re állítja. A méretek relatív változtatására, valamint az előző értékhez való visszatérésre vonatkozó szabályok azonosak az \s és a \f-re érvényes szabályokkal.

Makrók definiálása

A makrodefiniálás feladatát teljes általánosságban kezelve túlságosan belebonyolódnánk a **troff** rejtelmeibe, de néhány alapgondolatot anélkül is be tudunk mutatni. Példának következzék a **.CW** definíciója:

.de CW	Definíció kezdete.
\&\f(CW\\$1\fp\\$2	Betűtípusváltás az első argumentum körül.
..	Definíció vége.

A \$n az **n**-edik argumentum értékét jelenti a makró hívásakor; üres, ha nem volt **n**-edik argumentum. Az egymást követő két fordított perjel (\\) a makródefiniálás idején elhalasztja a **\$n** kiértékelését. A \& megvédi az argumentumot attól, hogy a **troff** parancsként értelmezze, ha esetleg ponttal kezdődik, mint a következő példában:

.CW .sp

9.3. A tbl és az eqn preprocesszorok

A **troff** egy kívül-belül nagy és bonyolult program, tehát módosítani, új funkcióval kiegészíteni egyáltalán nem könnyű. Ennek megfelelően a matematikai kifejezések és a táblázatok kiszedését támogató programokat más megközelítésben fejlesztették ki. Külön nyelveket terveztek, amelyeket az **eqn** és a **tbl** külön programok valósítanak

meg. Ezek a **troff** preprocesszoraiként működnek, amelyek a **troff**-ot a betűszedőgép assembly nyelvének tekintve erre a nyelvre fordítanak.

Az **eqn** a korábbi. Ez volt az **yacc** első, nem programnyelv fejlesztését célzó alkalmazása.* A **tbl** a későbbi, az **eqn**-hez hasonló szellemben, de merőben más szintaxissal készült. A **tbl** nyelvtana elég egyszerű, nem érdemes használni a **yacc**-ot.

A UNIX cső lehetősége erőteljesen sugalmazza a feladatok különálló programokra bontását. A feladat részekre bontásán kívül (ami elkerülhetetlen volt, hiszen a **troff** már önmagában közel akkora volt, amilyen nagy lehet egy program a PDP-11-en) a cső le is egyszerűsíti a részek és az érintett programozók közötti információcserét. Ez utóbbi különösen nagy jelentőségű – nem kell forráskód ahhoz, hogy egy preprocesszort készítsünk. Továbbá a csöveknek köszönhetően nem kell nyugtalankodnunk az óriási közbülső fájlok miatt! (Feltéve, hogy nem futtatjuk az összetevőket nyomkövetés céljából szándékosan különállóan.)

A különálló programok csöveken keresztül való egymáshoz kapcsolódása nehézségeket is okoz. A sebesség valamivel csökken, hiszen nagyon sok a be- és kimenet: mind az **eqn**, mind a **tbl** a bemenetének közel a nyolcszorosát írja a kimenetére. Nagyobb baj az, hogy az információ csak egy irányban áramlik. Nincs mód pl. arra, hogy az **eqn** meghatározza az éppen alkalmazott pontméretet, ami egy kicsit gyengíti a nyelvet. Végül a hibajelzés nehézkes, általában bonyolult egy **troff** hibaüzenetből az azt okozó **eqn** vagy **tbl**-beli hibára következtetni.

Mindazonáltal a szétválasztásból adódó haszon sokkal nagyobb, mint a hátrány, ezért ennek mintájára nagyon sok más preprocesszort is írtak.

Táblázatok

Kezdjük a **tbl** rövid elemzésével, ugyanis elsőként **hoc** dokumentációból a műveleti jelek táblázatát fogjuk bemutatni. A **tbl** beolvassa a bemeneti fájljait vagy a szabványos bemenetet, és a **.TS** (táblázat kezdete) és **.TE** (táblázat vége) parancsok közti szöveget **troff** parancsokká alakítja. Ezek kiírják a táblázatot, felsorakoztatják az oszlopokat, és az összes tipográfiai részletre ügyelnek. A **.TS** és a **.TE** a sorokat is átmásolja, így a makrocsomag megfelelően definiálhatja ezeket. Pl. biztosíthatja, hogy a táblázat egy lapra kerüljön és elkülönüljön az azt körülvevő szövegtől.

Annak ellenére, hogy bonyolultabb táblázatok elkészítéséhez meg kell nézni a **tbl** kézikönyvet, egy példa segítségével be lehet mutatni a legfontosabb elemeket. Íme a **hoc** dokumentáció egy részlete:

```
.TS
center, box;
c s
lfCW l.
\fB1. táblázat: \fp Operátorok csökkenő precedenciarendben
.sp .5
```

* Valószínűtlen, hogy az **eqn** elkészült volna, ha az **yacc** nem állt volna rendelkezésünkre.

```
                hatványozás (\s-1FORTRAN\s0 **), jobbról asszociatív
! \-            (egyváltozós) logikai és aritmetikai tagadás
* /            szorzás, osztás
+ \-            összeadás, kivonás
>>=             relációs jelek: nagyobb, nagyobb egyenlő,
<<=                             kisebb, kisebb egyenlő
\&== !-         egyenlő, nem egyenlő (mind azonos precedenciával)
&&              logikai ÉS (mindkét operandust mindig kiszámítja)
||              logikai VAGY (mindkét operandust mindig kiszámítja)
\&=             értékadás, jobbról asszociatív
.TE
```

ami a következő táblázatot állítja elő:

1. táblázat
Operátorok csökkenő precedenciarendben

^	hatványozás (FORTRAN-ban **), jobbról asszociatív
! -	(egyváltozós) logikai és aritmetikai tagadás
* /	szorzás, osztás
+ -	összeadás, kivonás
> >=	relációs jelek: nagyobb, nagyobb vagy egyenlő
< <=	kisebb, kisebb vagy egyenlő
== !=	egyenlő, nem egyenlő (mind azonos precedenciával)
&&	logikai És (mindkét operandust mindig kiszámítja)
||	logikai VAGY mindkét operandust mindig kiszámítja
=	értékadás, jobbról asszociatív

A TS parancs és a pontosvessző között álló szavak (**center**, **box**) a táblázat általános tulajdonságait írják le: a táblázatot a lapon vízszintes irányban középre kell állítani és be kell keretezni. Más lehetőségek is lennének, többek között a **doublebox**, az **allbox** (minden egyes táblázatelemet bekereteztünk), és az **expand** (a táblázatot a teljes lapszélességben széthúzzuk).

A táblázat különféle részleteinek, ez esetben a táblázat fejlécének és törzsének formáját a pontig bezárólag a következő sorok írják le. Az első a táblázat első sorára, a második a második sorra, az utolsó az összes további sorra vonatkozik. A példatáblázatok közül az 1. táblázatban csak két ilyen specifikációs sor van, így a második specifikáció a táblázat első utáni összes sorára érvényes. A formátumleíró jelek a következők: **c** a táblázatelemeket az oszlopok közepére állítja, **l** a bal szélre, **r** a jobb szélre igazít és **n** a számokat tizedespont szerint oszlopba rendezi, az **s** egy *eltolt* (spanned) oszlopot jelent. Az általunk használt **c s** azt jelenti, hogy a címet vízszintesen középre kell állítani, eltolva mind az első, mind a második oszlopot. A betűtípust egy oszlopra is meg lehet adni; az **lfCW** parancs egy bal szélhez igazított **CW** betűtípusú oszlopot jelent.

A formátumleírást a táblázat szövege követi. Az oszlopokat tabulátor karakterek és bizonyos, a táblázatok belsejében is ható **troff** parancsok (mint pl. **.sp**) választják el. (Vegyük észre a néhányszor előforduló **\&** jeleket, amelyek itt az a − és az = jeleket védik. Ezeket ugyanis a **tbl** különben úgy értelmezné, hogy a táblázatban ott egy vízszintes elválasztóvonalat kell húznia.)

A **tbl** sokkal többféle táblázatot tud előállítani, mint amilyeneket ez az egyszerű példa sejtet. Képes a szövegeket a táblázat rácsos szerkezetébe illeszteni, függőlegesen beállítani az oszlopok címeit stb. Bonyolultabb táblázatok használatához legegyszerűbb a *UNIX kézikönyv 2A* kötetének hasonló példáit megkeresni és átdolgozni.

Matematikai képletek

A **troff** másik preprocesszora, az **eqn** egy matematikai képleteket leíró nyelvet fordít **troff** parancsokra, kinyomtathatóvá téve így a képleteket. Automatikusan kezeli a betűtípus- és betűméret-változtatásokat, és lehetővé teszi, hogy a szabványos matematikai jelekre a neveikkel hivatkozzunk. Az **eqn** bemenete általában a **.EQ** és a **.EN** sorok közé kerül, a **tbl**-nél megismert **.TS** és **.TE**-hez hasonlóan. Pl. a

```
.EQ
x sub i
.EN
```

az x_i-t írja ki. Ha az **ms** makrocsomagot használjuk, akkor a képletet „kivetítésként" nyomtatjuk ki. A **.EQ** parancsban megadhatunk egy opcionális argumentumot is, amely a képlet számát adja meg. Pl. az

$$f(\zeta) = \frac{1}{2\pi i} \int_C \frac{f(z)}{z-\zeta} dz \qquad (9.1)$$

Cauchy-függvény integrálját így írjuk le:

```
.EQ (9.1)
f( zeta ) ~=~ 1 over {2 pi i} int from C
  f(z) over {z - zeta} dz
.EN
```

Az **eqn** nyelv a matematikai képletek olvasását követi. Az olvasott képletek és az **eqn** bemenete között az egyik eltérés, hogy a ({}) kapcsos zárójelek az **eqn** számára jelentenek zárójeleket – felülbírálják a nyelv alapértelmezés szerinti precedenciájára vonatkozó szabályokat –, míg a közönséges zárójeleknek semmilyen különös jelentősége sincs. A szóközöknek viszont *van* jelentősége! Figyeljük meg, hogy a fenti példában az első **dzeta**-t szóközök veszik körül, a kulcsszavakat, mint amilyen a **dzeta** vagy az **over** az **eqn** csak akkor képes felismerni, ha szóközök vagy kapcsos zárójelek fogják körül ezeket, amelyek a kimeneten már nem jelennek meg. Ha a kimeneten mindenképp meg akarunk jeleníteni egy szóközt, akkor egy hullámjelet (~), a kapcsos zárójelekhez pedig a "{"-t és a "}"-t kell alkalmaznunk.

Az **eqn** kulcsszavai több osztályba sorolhatók. A görög betűket kiejtés szerint írjuk mind kis-, mind nagybetűvel; így pl. **lambda** és **LAMBDA** jelentése λ és Λ. Más matematikai jeleknek nevük van, pl. ilyenek a **sum, int, infinity, grad,** amelyek jelentése rendre: Σ, $\int$, ∞, ∇. Vannak pozicionális operátorok, ilyenek a **sub, sup, from, to** és az **over.**

$$\sum_{i=0}^{\infty} x_i^2 \rightarrow \frac{1}{2\pi}.$$

A képlet leírása:

```
sum from i=0 to infinity x sub i sup 2 ~->~ 1 over {2 pi}
```

Ezenkívül más műveleti jelek is vannak. Pl. az **sqrt** és a nyújtott zárójelek. Az **eqn** oszlopokat és mátrixokat is össze tud állítani. Parancsok segítségével állítható a jelek mérete, típusa, elhelyezése is, ha nem megfelelő az alapértelmezés.

Egyszerű matematikai kifejezéseket, mint amilyen a $\log_{10}(x)$, nem kivetíteni szokás, hanem a szöveg közé helyezni. Az **eqn delim** kulcsszavával megadható egy karakterpár, amely a sorközi kifejezések közrezárására szolgál. A jobb és a bal oldali határt jelző karakterek általában azonosak; gyakran mindkettő a $ jel. A **hoc** azonban az argumentumokhoz használja a $-t, ezért mi a példáinkban a @-t alkalmazzuk. A % szintén használható határolókarakter, de a többit kerüljük; a különféle programokban ugyanis olyan sok karakternek van különleges jelentése, hogy igen látványos rendellenes viselkedést érhetünk el. (E fejezet írásakor nekünk is sikerült!) Így a

```
.EQ
delim @@
.EN
```

utáni sorközi kifejezéseket, mint amilyen a $\sum_{i=0}^{\infty} x_i$, a következőképpen lehet nyomtatni:

```
sorközi kifejezéseket,
mint amilyen a @ sum from i=0 to infinity x sub i@, a következőképpen lehet nyomtatni:
```

Sorközi kifejezéseket használunk a táblázatok matematikai képleteihez, ahogyan ezt a **hoc** leírásából vett példa mutatja:

```
.TS
center, box
c s s
lfCW n l.
\fB3. táblázat: \fP                    Beépített állandók
.sp .5
DEG     57.29577951308232087680        @ 180/ pi@mfok/radián
E       2.71828182845904523536         @e@, természetes logaritmus alapszáma
GAMMA   0.57721466490153286060         @gamma@, Euler–Mascheroni-állandó
PHI     1.61803398874989484820         @(sqrt 5+1)/2@, az aranymetszés aránya
PI      3.14159265358979323846         @pi@, végtelen nemperiodikus tizedestört
.TE
```

Ez a példa azt is mutatja, hogyan rendezi a **tbl** tizedespont szerint oszlopba a számokat. Az eredmény a következő példatáblázat:

3. táblázat

Beépített állandók

DEG	57,29577951308232087680	$180/\pi$, fok/radián
E	2,71828182845904523536	e, természetes logaritmus alapszáma
GAMMA	0,57721566490153286060	γ, Euler – Mascheroni-állandó
PHI	1,61803398874989484820	$(\sqrt{5}+1)/2$, az aranymetszés aránya
PI	3,14159265358979323846	π, végtelen nemperiodikus tizedestört

Végül megemlítjük, hogy az **eqn** minden olyan szöveget, amit nem ismer fel, *dőlt* betűvel nyomtat ki. Ezért elterjedt szokás a közönséges szavakat dőlt betűsen az **eqn** segítségével íratni. Pl. a *@Szó@* hatására az **eqn** ezt írja: *Szó*. Figyeljünk arra, hogy az **eqn** néhány hétköznapi angol szót kulcsszóként ismer (ilyen a **from** és a **to**) és különlegesen kezel, azonkívül a szóközöket lenyeli, így a dőlt betűs trükköt óvatosan kell használni!

Kiíratás

Ha egyszer elkészült a dokumentumunk, akkor ahhoz, hogy kiírhassuk, végig kell futtatnunk az összes preprocesszoron, majd a **troff**-on. A sorrend: **tbl, eqn, troff.** Ha rögtön a **troff**-ot indítjuk, akkor a

```
$ troff -ms fájlnevek      (Vagy: -mm)
```

parancsot adjuk ki. Ha nem, akkor a csőhálózat első parancsának argumentumaként kell a fájlneveket megadni, a többi programnak pedig szabványos bemenetét kell olvasnia úgy, mint

```
$ eqn fájlnevek | troff -ms
```

vagy

```
$ tbl fájlnevek | eqn | troff -ms
```

Kényelmetlen azzal foglalkozni, hogy ténylegesen melyik preprocesszor szükséges az adott dokumentum kinyomtatásához. Úgy gondoltuk, érdemes egy olyan programot írni, amelyik összeállítja a megfelelő parancssort. Ezt **doctype**-nek neveztük el:

```
$ doctype sz9.*
cat sz9.1 sz9.2 sz9.3 sz9.4 | pic | tbl | eqn | troff -ms
$ doctype hoc.ms
cat hoc.ms | tbl | eqn | troff -ms
$
```

A **doctype**-ot a negyedik fejezetben részletesen tárgyalt segédprogramokra építve valósítottuk meg; egy **awk** program megkeresi a preprocesszorok által használt parancssorozatokat és kiírja parancssorba azokat a programokat, amelyek szükségesek a dokumentum kialakításához. A **.PP** (bekezdés kezdete) parancsot is figyeli, amelyet az **ms** formakialakító makrocsomagot használ.

```
$ cat doctype
# doctype: a megfelelő troff parancssor összeállítása
echo -n "cat $* | "
egrep -h '^\.(EQ|TS|\[|PS|IS|PP)' $* |
sort -u |
awk '
/^\.PP/          { ms++ }
/^\.EQ/          { eqn++ }
/^\.TS/          { tbl++ }
/^\.PS/          { pic++ }
/^\.IS/          { ideal++ }
/^\.\[/ .        { refer++ }
END {
```

```
        if (refer >0)       printf "refer | "
        if (pic >0)         printf "pic | "
        if (ideal >0)       printf "ideal | "
        if (tbl >0)         printf "tbl | "
        if (eqn >0)         printf "eqn | "
        printf "troff "
        if (ms >0) printf "-ms"
        printf "\n"
}'
$
```

(Az **egrep** a **-h** kiegészítéssel a sorok elején nem írja ki a fájlnévfejléceket, sajnos ez az opció nem minden rendszerváltozatban működik.) A bemenetet végigvizsgálva összegyűlik, hogy milyen összetevőket használunk. A teljes bemenet megvizsgálása után a megfelelő sorrendben elvégezve a feldolgozást kinyomtatjuk a kimenetet. A részletek csak azt mutatják, hogyan lehet a **troff** dokumentumokat a szabványos preprocesszorokkal formázni, az alapgondolat azonban általános érvényű: bízzuk a részleteket a gépre!

A **doctype**, akárcsak a **bundle**, példa egy programot előállító programra. Jelenlegi formájában felhasználójának mindig újra be kell gépelnie a parancssort ahhoz, hogy a burok végrehajtsa. Ennek feloldása az egyik gyakorló feladatra marad.

Amikor már a **troff** parancsokat futtatjuk, nem szabad elfeledkeznünk arról, hogy a **troff** működése rendszerfüggő: bizonyos rendszerekben közvetlenül vezérli a szedőgépet, míg másutt a szabványos kimenetén előállított adatait egy külön programnak kell elküldenie a szedőgépre.

Megemlítjük, hogy e program első változata nem használta sem az **egrep**-et, sem a **sort**-ot, az **awk** maga nézte végig a teljes bemenetet. Ez nagyobb dokumentumok esetén túl lassúnak bizonyult, ezért kiegészítettük az **egrep**-pel, ami gyors keresést végez, majd a **sort -u**-val, ami az ismétléseket szűri ki. Egy átlagos dokumentum esetén a két kiegészítő adatszűrő program jelentette terhelés kisebb, mint ha az **awk**-ot a teljes bemeneten futtatnánk. Ennek szemléltetésére összehasonlítjuk a doctype-ot és egy, csak az **awk**-ot futtató változatot. (Mindkettővel e fejezet eredeti angol szövegének kb. 52000 karakterét dolgozták fel.)

```
$ time awk '... doctype egrep nélkül ...' sz9.*
cat sz9.1 sz9.2 sz9.3 sz9.4 | pic | tbl | eqn | troff -ms

real        31.0
user         8.9
sys          2.8
$ time doctype sz9.*
cat sz9.1 sz9.2 sz9.3 sz9.4 | pic | tbl | eqn | troff -ms

real         7.0
user         1.0
sys          2.3
$
```

Az összehasonlítás eredménye egyértelműen a három eljárásra építő változatnak kedvez. (Ezt egy olyan gépen mértük, ahol nem volt más felhasználó; az átfutási idő (real time) aránya egy leterhelt rendszerben még inkább az **egrep**-es változatnak kedvezne.) Vegyük észre, hogy ismét először egy egyszerű változat készült el, és csak azután kezdtük a hatékonyságot javítani!

9-2. feladat. Miképp alakíthatnánk ki e fejezet külalakját? □

9-3. feladat. Ha éppen a dollár az **eqn** határolójele, akkor hogyan lehet egy dollárjelet a kimeneten megjelentetni? Útmutató: nézzük át az idézőjeleket és az **eqn** kulcsszavait! □

9-4. feladat. Miért nem működik a

```
$ `doctype fájlnevek`
```

parancs? Módosítsuk a **doctype**-ot úgy, hogy lefuttassa az összeállított parancsot ahelyett, hogy kinyomtatná! □

9-5. feladat. Jelentős-e az a terhelés, amit a **doctype**-ban a kiegészítő **cat** parancs okoz? Írjuk át úgy a **doctype**-ot, hogy ez ne legyen benne! Melyik változat az egyszerűbb? □

9-6. feladat. Melyik jobb: a **doctype**-ot használni, vagy egy burokfájlt írni, amelyik az adott dokumentumhoz szükséges parancsokat tartalmazza? □

9-7. feladat. Kísérletezéssel állítsuk elő a lehető leggyorsabb **doctype**-változatot a **grep, egrep, fgrep, sed, awk** és **sort** különböző összeállításaiból! □

9.4. A kézikönyvlap

Egy parancs fő dokumentálása általában a *UNIX kézikönyv* számára készült egyoldalas leírás, a kézikönyvlap (l. a 9.2. ábrát). A kézikönyv anyagát egy szabványos jegyzék, általában a **/usr/man** alatt tartják, a kézikönyv szakaszai szerint számozott aljegyzékben. A mi **hoc** kézikönyvlapunk pl. a **/usr/man/man1/hoc.1** alatt található, hiszen ez egy felhasználói parancsot ír le.

A kézikönyvlapokat a **man(1)** paranccsal nyomtatjuk ki. Ez egy **nroff −man** parancsot futtató burokfájl, így a **man hoc** a **hoc** kézikönyvet nyomtatja ki. Ha ugyanaz a név több részben is szerepel, mint ahogy ez magára a **man**-ra is igaz (az első rész írja le a parancsot, míg a hetedik rész a makrókat), akkor a megfelelő rész megadható a **man** parancsban. A következő parancs pl. csak a makrók leírását írja ki:

```
$ man 7 man
```

Alapértelmezés szerint a parancs az összes, a megadott névre vonatkozó lapot kinyomtatja az **nroff**-fal, de a **man**-t a **troff**-ra építve nyomdakész lapokat állít elő.

A kézikönyvlap szerzője a **/usr/man** megfelelő aljegyzékben létrehoz egy fájlt. A **man** parancs a lap kinyomtatásához egy makrocsomaggal együtt hívja meg az **nroff**-ot avagy a **troff**-ot. Ezt magunk is láthatnánk, ha végignéznénk a **man** parancs formázási hívásait. Eredményünk ez lenne:

```
$ grep roff `which man`
nroff $opt -man $all ;;
neqn $all | nroff $opt -man ;;
troff $opt -man $all ;;
troff -t $opt -man $all | tc ;;
eqn $all | troff $opt -man ;;
eqn $all | troff -t $opt -man | tc ;;
$
```

Ez a nagy változatosság a választási lehetőségek sokaságát tárja elénk: **nroff**-ot vagy **troff**-ot használjunk, futtassuk vagy ne futtassuk az **eqn**-t stb. A kézikönyv makrocsomag olyan **troff** parancsokat ír le, amelyek a kézikönyv stílusában szerkesztik meg a leírást. A **troff -man** ezeket hívja. A makrocsomag alapjában megegyezik az **ms** makrókkal, de eltérések is vannak, különösen a cím kiszedésében, valamint a betűtípusváltó parancsokban. A makrókat leírja a **man(7)**, de az alapvető dolgokat könnyű megjegyezni. A kézikönyvlap vázlata a következő:

```
.TH COMMAND fejezetszám
.SH NAME
Parancs \- a funkció rövid leírása
.SH SYNOPSIS
.B parancs
opciók
.SH DESCRIPTION
A programok és opcióik részletes leírása.
A bekezdéseket .PP vezeti be.
.PP
Ez egy új bekezdés
.SH FILES
A parancs által használt fájlok, pl. passwd(1) említi a /etc/passwd fájlt
.SH "SEE ALSO"
A kapcsolódó leírásokra való hivatkozások, ide értve más kézikönyvlapokat is.
.SH DIAGNOSTICS
A rendellenes kimenet leírása (pl. l. cmp(1))
.SH BUGS
Meglepő sajátosságok (nem feltétlenül hibák, l. később)
```

Ha valamelyik rész üres, akkor annak a fejlécét is elhagyjuk. A **.TH** sor, valamint a **NAME, SYNOPSIS** és **DESCRIPTION** részek kötelezőek! A

```
.TH COMMAND fejezetszám
```

sor megnevezi a parancsot és megadja a fejezet számát. A különféle **.SH** sorok azonosítják a kézikönyvlap szakaszait. A **NAME** és **SYNOPSIS** szakaszok különlegesek; a többi egyszerű. A **NAME** szakasz megnevezi a *parancsot* (ezúttal kisbetűvel), és egy egysoros leírást ad. A **SYNOPSIS** rész megnevezi az opciókat, de nem írja le azokat. Mint a többi részben, ebben is a bemenet szabad formátumú; a betűtípusváltásokat a **.B, .I** és **.R** makrókkal adhatjuk meg. A **SYNOPSIS** részben a parancs neve és az opciók kövér betűsek, míg a leírás többi része normál betűs. Az **ed(1) NAME** és **SYNOPSIS** szakasza pl. a következő:

```
.SH NAME
ed \- szövegszerkesztő
.SH SYNOPSIS
.B ed
[
.B \-
] [
.B \-x
] [ név ]
```

Ez a következőképpen íródik ki:

NAME
ed – szövegszerkesztő
SYNOPSIS
ed [–] [–x] [név]

Vegyük észre, hogy a \– jelet használtuk, nem pedig az egyszerű (–) kötőjelet!

A **DESCRIPTION** rész a parancsokat és azok opcióit írja le. A legtöbb esetben ez csak magával a paranccsal foglalkozik, és nem tér ki arra a nyelvre, amelyet a parancs meghatároz. A **cc(1)** kézikönyvlap nem a C nyelv leírása; hanem azt mondja meg, hogyan kell a **cc** paranccsal C nyelvű programokat lefordítani, hogyan hívjuk meg az optimalizálót, hová kerül a kimenet stb. A nyelv leírása a C kézikönyvben található és a **cc(1)**-nek a SEE ALSO részében erre hivatkozunk is. Ez az elv azonban nem abszolút érvényű: a **man(7)** a kézikönyv makróinak nyelvét írja le.

A **DESCRIPTION** rész hagyományosan dőlt betűket használ a parancsnevek és a kiegészítő opciók (mint az **ed** lapján a név) kiírására. A **.I** (az első argumentum dőlt betűs) és a **.IR** (az első argumentum dőlt betűs, a második álló betűs) makrók a dőlt betűs kiírást nagyon megkönnyítik. A **.IR** makró azért szerepel itt, mivel a **man** csomag **.I** makrója nem azonos az **ms**-ével, nem követi a második argumentum dokumentálatlan, de kényelmes kezelését.

A **FILE** rész felsorolja az összes, a parancs által akár közvetve használt fájlt. A **DIAGNOSTICS**-ot csak akkor kell megadni, ha a parancs rendhagyó üzenetekkel is rendelkezik. Ezek hibaüzenetek, kilépési státusok vagy a parancs normális viselkedésének meglepő változatai lehetnek.

A **BUGS** rész elnevezése is egy kicsit megtévesztő. Az itt közöltek nem valódi hibák, inkább tökéletlenségek – az egyszerű hibákat a parancs beépítése előtt ki kell javítani. Ahhoz, hogy legyen valami benyomásunk arról, mi is kerül a **DIAGNOSTICS** és a **BUG** részekbe, elég végiglapozni a *UNIX kézikönyvet*.

Egy példa megvilágíthatja, hogyan kell megírni egy kézikönyvlapot. A **hoc(1)** leírás forrását, a **/usr/man/man1/hoc.1**-et a következőkben mutatjuk be:

```
.TH HOC 1
.SH NAME
hoc \- interaktív lebegőpontos nyelv
.SH SYNOPSIS
.B hoc
[ file ... ]
.SH DESCRIPTION
A
.I hoc
egy egyszerű, BASIC szintű
lebegőpontos aritmetikai nyelvet
értelmez, C nyelv jellegű szintaktikával,
függvények és eljárások argumentumaival
és rekurzióval.
.PP
A megnevezett
.IR fájlokat
sorban beolvassa és értelmezi.
Ha nincs
.I fájl
megadva, vagy ha a
.I fájl
a `\-'
akkor
.I hoc
a szabványos bemenetet értelmezi.
.PP
A
.I hoc
bemenete
.I kifejezésekből
és
.IR utasításokból
áll.
A kifejezések kiértékelődnek és az
értékük kiíródik. Az utasítások, a
szimbolikus értékadások és a függvények
vagy az eljárásdefiníciók kimenete
nem íródik ki, hacsak nem hívják meg
a
.IR print-et.
.SH "SEE ALSO"
```

```
.I
Hoc \— Interaktív lebegőpontos
aritmetikai nyelv, írta Brian Kernighan és Rob Pike.
.br
.IR bas (1),
.IR bc (1)
és
.IR dc (1).
.SH BUGS
A függvények és eljárások definícióiban
a hibajavítás nem tökéletes.
.br
Az újsor-karakterek kezelése nem igazán kényelmes.
```

A következő parancs, a

```
$ man -t hoc
```

kimenete:

HOC(1) HOC(1)

NAME

hoc — interaktív lebegőpontos nyelv

SYNOPSIS

hoc [file...]

DESCRIPTION

A *hoc* egy egyszerű, BASIC szintű lebegőpontos aritmetikai nyelvet értelmez, C nyelv jellegű szintaktikával, függvények és eljárások argumentumaival és rekurzióval.

A megnevezett *fájlokat* sorban beolvassa és értelmezi. Ha nincs megadva *fájl* vagy ha a fájl —, akkor a *hoc* a szabványos bemenetet értelmezi.

A *hoc* bemenete *kifejezésekből* és *utasításokból* áll. A kifejezések kiértékelődnek, és az értékük kiíródik. Az utasítások, a szimbolikus értékadások és a függvények vagy az eljárásdefiníciók kimenete nem íródik ki, hacsak nem hívják meg a *print*-et.

SEE ALSO

Hoc — Interaktív Lebegőpontos Aritmetikai Nyelv, írta Brian Kernighan és Rob Pike.
bas(1), *bc*(1) és *dc*(1).

BUGS

A függvények és eljárások definícióiban a hibajavítás nem tökéletes.
Az újsor-karakterek kezelése nem igazán kényelmes.

8. kiadás

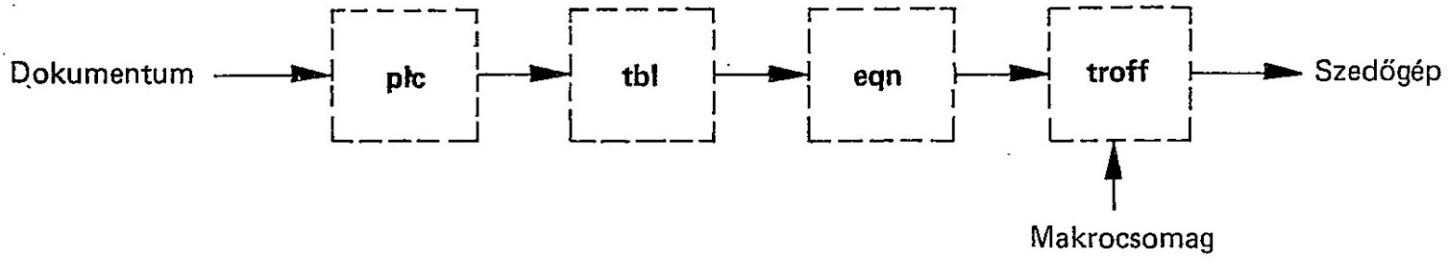
9-8. feladat. Írjunk a **doctype**-hoz kézikönyvlapot! Írjuk meg a **man** parancs egy olyan változatát, ami a saját **man** jegyzékünkben keresi saját programjaink leírását. □

9.5. Más dokumentumkészítő eszközök

Több más, a dokumentumkészítést támogató program is ismert. A **refer(1)** parancs kulcsszavakra hivatkozásokat keres, elhelyezi a szövegközti idézeteket az anyagunkban, és felépíti az anyag végén a hivatkozási részt. Megfelelő makrók létrehozásával elérhetjük, hogy a **refer** a hivatkozásokat az általunk megkívánt különleges stílusban nyomtassa ki. A sokféle számítástudományi szaklap igényéhez van már ilyen makró. A **refer** szerepel a UNIX 7. kiadásában, de a többi változat némelyikéből hiányzik.

A **pic(1)** és az **ideal(1)** ábrákat készítenek, hasonlóan ahhoz, ahogy az **eqn** matematikai képleteket. Az ábrák jelentősen bonyolultabbak az egyenleteknél (legalábbis a nyomtatás szempontjából), és nincs semmilyen szájhagyománya annak, hogyan nevezzük meg az ábrák részleteit. Így mindkét nyelv megtanulásához és használatához egy kis erőfeszítésre van szükség. A **pic** szemléltetésére bemutatunk egy egyszerű ábrát, és annak **pic**-nyelvű leírását:

```
.PS
.ps −1
box invis "dokumentum"; arrow
box dashed "pic"; arrow
box dashed "tbl"; arrow
box dashed "eqn"; arrow
box "troff"; arrow
box invis "szedőgép"
[ box invis "makro" "csomag"
  spline right then up −>] with .ne at 2nd las box.s
.ps +1
.PE
```



A **pic** és az **ideal** nem szerepel a UNIX 7. kiadásában, de ma már hozzáférhetőek.

Mind a **refer**, mind a **pic**, mind az **ideal** a **troff** preprocesszorai. Ezeken kívül olyan programok is vannak, amelyek dokumentumaink szövegének vizsgálatára és értékelésére szolgálnak. Ezek közül a legismertebb a **spell(1)**, amelyik a valószínűsíthető helyesírási hibákra figyelmeztet. A **style(1)** és a **diction(1)** a központozást, a nyelvtant és a nyelvhasználatot elemzi. Ezekből fejlődött ki a „Writer's Workbench"

(az *Írói Munkaasztal*), azaz az írások stílusának javítását támogató programok rendszere. Ez jól használható a túl sokat használt kifejezések, felesleges szavak, rossz kifejezések kiszűrésére.

A **spell** szabványos program. Olvasóink UNIX rendszereiben más programok is lehetnek, a **man**-nal ez könnyen kideríthető:

```
$ man style diction wwb
```

vagy a **/bin** és a **/usr/bin** jegyzékben levő fájlokat is kilistázhatjuk.

Történeti és irodalmi megjegyzések

A **troff**-nak, amelyet a néhai Joe Ossanna írt a Graphics System CAT-4 szedőgéphez, hosszú előtörténete van. Egészen a **RUNOFF**-ig nyúlik vissza, amelyet J. E. Saltzer írt a CTSS-re a MIT-nél, a hatvanas évek elején. E programoknak közös az alapgondolata, és közös az alapvető parancsok szintaktikája is. A **troff** azonban a leghatékonyabb és legbonyolultabb, és az **eqn** meg a többi preprocesszor jelentősen fokozza használhatóságát. Több újabb, kulturáltabb bemenetű szövegformázó program is létezik; ezek közül a legismertebbek valószínűleg a Don Knuth-féle **TEX** (*TEX and Metafont: New Directions in Typesetting,* Digital Press, 1979), és Brian Reid Scribe-ja („Scribe: a high level approach to computer document formatting", 7th Symposium on the Principles of Programming Languages, 1980). Ezt a területet jól áttekinti Richard Furuta, Jeffrey Scofield és Alan Shaw cikke: „Document Formatting Systems: Survey, Concepts and Issues" (Computing Surveys, 1982, szeptember).

Az **eqn**-ről szóló első publikáció az „A system for typesetting mathematics" (*CACM,* 1975. március); szerző Brian Kernighan és Lorinda Cherry. Az **ms** makrocsomagot, a **tbl**-t és **refer**-t mind Mike Lesk készítette. Ezek leírása csak a *UNIX kézikönyv 2A* kötetében található.

A **pic**-et Brian Kernighan cikke, „PIC – a language for typesetting graphics" (Software – Practice and Experience, 1982. január), az **ideal**-t Chris Van Wyk cikke, „A high-level language for describing pictures" (*ACM Transactions on Graphics,* 1982. április) írja le.

A **spell** parancs Steve Johnson burokfájljából Doug McIlroy munkájával alakult C nyelvű programmá. A UNIX 7. kiadásában szereplő **spell** egy „hashing" módszert használ a gyors átnézéshez, alkalmaz továbbá a szótár méretének csökkentésére a szavakról az elő- és az utótagokat, ragokat, képzőket automatikusan leválasztó szabályokat. Erről szól M. D. McIlroy cikke: „Development of a spelling list", *IEEE Transactions on Communication,* 1982. január.

A **style** és a **diction** programokat Lorinda Cherry cikke írja le: „Computer aids for writers", SIGPLAN Symposium on Text Manipulation, Portland, Oregon (1981. június).

10. Epilógus

A UNIX operációs rendszer már régen elmúlt tízéves, mégis egyre gyorsabban növekszik az azt futtató számítógépek száma. Egyedülálló sikere ez egy nem üzleti céllal tervezett rendszernek.

A kereskedelmi siker oka valószínűleg a nagyarányú hordozhatóság – az a tulajdonság, hogy a rendszer a fordítók és a mag egyes részeinek kivételével bármilyen számítógépen változatlanul fut. Így a UNIX szoftvert használók viszonylag kevés munkával tehetik át megszokott rendszerüket új hardverre, és felhasználhatják a UNIX programok bővülő kínálatát is.

A UNIX rendszer azonban már kereskedelmi sikere előtt is népszerű volt, már akkor is, amikor még csak PDP-11 számítógépen futott. Ritchie és Thompson 1974-es CACM tanulmánya felkeltette a szakemberek érdeklődését, és 1975-re az egyetemeken általánosan elterjedtek a 6. kiadású rendszerek. Az 1970-es évek közepéig a UNIX ismeretek szájhagyomány útján terjedtek: a rendszert minden támogatás és garancia nélkül kapta meg felhasználója. Alkalmazói mégis annyira lelkesültek érte, hogy másokat is rábeszéltek, és aki kipróbálta, általában megmaradt mellette. Mai sikerének egyik fontos tényezője, hogy az egyetemi UNIX rendszerek mellett nevelkedett programozói generáció a munkahelyén is UNIX környezetet vár.

Miért lett annyira népszerű? A legfontosabb tényező, hogy kevés (mindössze kettő) és kivételes tehetségű ember tervezte és készítette. Céljuk tisztán az volt, hogy maguknak megfelelő programfejlesztési környezetet alkossanak, és szabadon meg is valósíthatták elképzeléseiket. A piac nyomásától mentes korai rendszerek olyan kicsik voltak, hogy egyetlen ember áttekinthette, megérthette az egészet. John Lions a 6. kiadás magját az ausztráliai New South Wales egyetemén, egy középfokú operációs rendszerek kurzuson tanította. Előadásaihoz írt jegyzetében ezt olvashatjuk: „...az egész dokumentáció elfér egy hallgató levéltárcájában." (Az újabb változatokban ezen már segítettek.)

E korai rendszer a számítástudományi találmányoknak – köztük az áradatfeldolgozás (csövek), a szabályos kifejezések, a nyelvelmélet (**yacc, lex** stb.) és olyan különlegességek, mint a **diff** algoritmusa – egész kincsesháza volt. Mindezeket összevéve olyan tulajdonságú mag keletkezett, amilyen „nagyobb operációs rendszerekben is ritkán található". Példaként tekintsük a be- és kimenet szerkezetét: hierarchikus fájlrendszer, ami akkoriban ritkaság volt; a fájlrendszerbe beépített eszközök, amelyekhez nem kellenek külön segédprogramok; néhány tucat kényes rendszerhívás,

köztük a két kritikus argumentummal rendelkező **open**. A magas szintű nyelven készült szoftver forráskódját a rendszerrel együtt terjesztették, így az tanulmányozható és módosítható volt.

A UNIX rendszer azóta a szoftverpiac állandó, bevált rendszerévé nőtte ki magát. A piaci igények eluralkodása miatt meg kellett komolyodnia, és bővülnie kellett a konkurens rendszerek által nyújtott „szolgáltatásokkal". Következmény: a mag az elmúlt tíz évben tízszeresére növekedett, de nem lett ugyanennyiszer jobb. Növekedését a létező környezetre nem építő, beteges programok elszaporodása kísérte. Nézni is rossz a programok eredeti szándékával ellentétes parancsopciók burjánzását. A forráskódot gyakran nem szállítják a rendszerrel együtt, egyre ritkább a jó programozási stílus.

Szerencsére azonban még a nagyobb változatokat is áthatják azok a gondolatok, amelyek a korai változatokat oly népszerűvé tették. Így a UNIX alapelvei – a szerkezetek egyszerűsége, az aránytalanul nagy eszközök hiánya, újraírás helyett a meglévő programokra való építés, a parancsértelmező programozhatósága, a fastruktúrájú fájlrendszer stb. – terjednek, kiszorítják a korábbi monolitikus rendszerszemléletet. A UNIX sem lesz örökéletű, de az azt felülmúlni kívánó rendszerek sok alapgondolatot át kell, hogy vegyenek belőle.

Az Előszóban elmondtuk, hogy a UNIX egy filozófia, a programozási feladat megközelítésének egy módszere. Visszapillantva a könyvre, Olvasóink bizonyára látják példáinkban ennek elemeit. A legfontosabbakat a következőkben összefoglaljuk.

Először, hagyjuk a gépet dolgozni! Azokra a feladatokra, amelyeket más rendszerekben kézi erővel kell megoldani, alkalmazzuk a **grep, wc, awk** programokat és társaikat.

Másodszor, hagyjuk a többieket dolgozni! Programjaink építőkockáiként használjuk fel a már meglévő programokat, a burok és a programozható szűrők segítségével fűzzük egybe azokat. Írjunk a munka lényegét elvégző, meglévő programhoz kis interfész programot, mint ahogyan az **idiff** esetében tettük. A UNIX környezet szokatlanul gazdag eszköztára milliófféleképpen kombinálható; a mi munkánk gyakran csupán a helyes kombináció kigondolása.

Harmadszor, bontsuk elemeire a feladatot! Készítsük el a legkisebb hasznos elemet, ennek tapasztalatai alapján döntsük el, hogy mi legyen a következő lépés (ha egyáltalán maradt még feladat). Ne bővítsük újabb tulajdonságokkal és opciókkal, amíg meg nem bizonyosodtunk azok szükségességéről.

Negyedszer, eszközöket építsünk! Olyan programokat írjunk, amelyek beépülnek a meglévő környezetbe, nem egyszerűen hozzátevődnek. Jól építkezve az ilyen programok a közös eszközkincs részeivé válhatnak.

Az Előszóban azt is mondtuk, hogy a rendszer nem tökéletes. Kilenc fejezet után, annyi különös megállapodás, érthetetlen különbözőség és önkényes korlátozás láttán Olvasóink bizonyára egyetértenek velünk. Mégis, az ilyen hibák ellenére is, az előnyök oldalára billen a mérleg, nem az ilyen zavaró kidolgozatlanságokéra. A UNIX rendszer valóban nagyon jó arra, amire tervezték: kényelmes programozási környezet.

Bár a UNIX kezd öregedni, még ma is életképes, és még mindig egyre népszerűbb. Ez a népszerűség arra vezethető vissza, hogy 1969-ben néhány világosan goldolkodó ember egy táblára olyan programozási környezet tervét vázolta fel, amelyet ők kényelmesnek találtak. Nem hitték, hogy rendszerük a számítógépek tízezreire fog eljutni, mégis egy egész programozói generáció hálás azért, hogy ez így történt.

1. függelék

Az editor összefoglalása

A UNIX „szabványos" szövegszerkesztője az **ed** nevű program, amelyet eredetileg Ken Thompson írt. Az **ed** programot az 1970-es évek elején kisszámítógépes környezetre (az első UNIX rendszerben a felhasználói programok legfeljebb 8 Kbyte méretűek lehettek), nagyon lassú (10 – 15 karakter/s) és papírra író terminálokra tervezték. Elődje egy akkoriban népszerű **qed** nevű szerkesztő volt.

A technológia fejlődése ellenére az **ed** majdnem változatlan maradt. Szinte biztos, hogy Olvasónk rendszerében van vonzóbb tulajdonságú szerkesztő is, pl. olyan képernyőszerkesztő, amely a változtatásokat azonnal mutatja.

Akkor miért töltsük időnket egy ilyen régimódi programmal? Azért, mert az **ed** hajlott kora ellenére néhány dolgot egészen jól csinál. Továbbá azért, mert minden UNIX rendszerben megtalálható; más gépre költözve biztosan kéznél lesz. Kis sebességű telefonvonalakon és bármilyen terminálon jól működik. Az **ed** könnyen futtatható fájlból érkező bemenettel, míg a legtöbb képernyőszerkesztő feltételezi, hogy egy terminált vezérel.

Az **ed** a minta szerinti keresésben szabályos kifejezéseket használ. Az **ed** szabályos kifejezései áthatják a UNIX rendszert: a **grep** és **sed** majdnem ugyanezeket használják; az **egrep**, **awk** és **lex** kiterjesztik ezeket; a burok a fájlnév-egyeztetésre más szintaxissal ugyanezeket az elveket használja. Némelyik képernyőszerkesztő „soronkénti módra" tud váltani, hogy az **ed**-hez hasonlóan kezelhesse a szabályos kifejezéseket.

Végül, az **ed** gyorsan fut. Könnyen elképzelhető, hogy az **ed**-et behívjuk egy egysoros változtatásra, az új változatot kiírjuk, kilépünk, és mindez kevesebb idő, mint az, amennyi alatt egy nagyobb, ügyesebb képernyőszerkesztő egyáltalán elindul.

Alapok

Az **ed** egyszerre egy fájlt szerkeszt. A fájlról készült másolaton dolgozik; a változtatásokat az eredeti fájlba explicit paranccsal rögzíthetjük. Az **ed**-nek vannak olyan parancsai, amelyekkel egymás utáni sorokat, mintával egyező sorokat kezelhetünk, és amelyekkel sorokat lehet változtatni.

Mindegyik **ed** parancs egy karakterből áll, amely rendszerint egy betű. A legtöbb parancs elé egy vagy két *sorszámot* írhatunk, amelyek jelzik, hogy a parancs mely

sorra vagy sorokra vonatkozzék; egyébként a parancs az alapértelmezés szerinti sorszámot használja. A sorszámok megadhatók a fájlban elfoglalt abszolút helyzettel (1, 2, ...), olyan rövidítésekkel, mint az utolsó sort jelentő $ vagy az aktuális sort jelentő pont (.), továbbá szabályos kifejezéseket alkalmazó mintakereséssel és ezek kombinálásával.

Tekintsük át újra, hogy az 1. fejezetben az **ed** felhasználásával hogyan hoztunk létre De Morgan verséből fájlokat!

```
$ ed vers
?vers                   Figyelmeztetés: a fájl nem létezik.
a                       Sorok hozzátétele kezdődik.
Nagy bolhák hátán kis bolhák:
  csípős bájt lelni ebben.
S még kisebb bolhák megtoldják
  e sort, mely végtelen.
.                       A hozzátételt a . lezárja
w vers                  A sorokat a vers fájlba írjuk
116                     ed jelenti, hogy 116 karaktert írt.
q                       Kilépés.
$
```

Az **a** parancs a fájlhoz sorokat hozzátesz vagy hozzáfűz; a hozzátétel módját a csak pontot (.) tartalmazó sor zárja le. Semmi sem jelzi, hogy éppen milyen módban vagyunk, így a két leggyakoribb hiba, hogy az **a** parancs nélkül elkezdünk szöveget beírni, és hogy a lezáró (.) sor bevitele előtt parancsot írunk be.

Az **ed** sosem írja automatikusan fájlba a szövegünket; a **w** paranccsal kell erre utasítanunk. Ha a változtatások kiírása nélkül próbálunk kilépni, akkor az **ed** figyelmeztetésképpen egy kérdőjelet (?) ír ki. Ekkor egy újabb **q** paranccsal a változtatások kiírása nélkül kiléphetünk. A **Q** parancs a változtatásoktól függetlenül mindig kilép.

```
$ ed vers
116                     A fájl létezik, 116 karaktert tartalmaz.
a                       Újabb sorokat teszünk a végéhez.
Nagy bolhák rendje még nagyobb
  bolhák rendjét elözi;
S még nagyobbakét ... tudhatod,
  jön még nagyobb, s a többi.
.                       Hozzátétel vége.
q                       Megpróbálunk kilépni.
?                       Figyelmeztetés: nem írtuk ki!
w                       Nincs fájlnév, a vers nevet feltételezi.
232
q                       Most már jó a kilépés!
$ wc vers               Biztonsági ellenőrzés.
    8  36  232 vers
$
```

Szökés felkiáltójellel a burokhoz

Ha az **ed**-et futtatjuk, akkor abból ideiglenesen kiszökhetünk egy másik burok parancs futtatására; ehhez nem kell az **ed**-ből kilépni. A **!** parancs ezt csinálja:

```
$ ed vers
232
!wc vers                    Az ed elhagyása nélkül futtatjuk a wc-t.
    8  36  232 vers
!                            Visszatértünk a parancsból.
q                            Nincs változtatás; jogos a w nélküli kilépés.
$
```

Kiíratás

A fájl sorai az 1, 2, ... sorszámokkal vannak megjelölve; az n-edik sort az **np** parancs vagy az n szám megadásával, az m-ediktől az n-edikig terjedő sorokat az **m,np** parancs megadásával írathatjuk ki. Az utolsó sor „sorszáma" a $ (tehát nem kell megszámlálni a sorokat).

```
1                    Kiírja az első sort; ugyanaz, mint 1p.
$                    Kiírja az utolsó sort; ugyanaz, mint $p.
1,$p                 Kiírja az összes sort.
```

A sorokat egyenként egyszerűen a *RETURN* megnyomásával írathatjuk ki; a mínusz-jellel (−) egy sort lehet visszalépni. A sorszámokat a + és − jelekkel kombinálhatjuk:

```
$−2,$p               Az utolsó három sort írja ki.
1,2+3p               Az 1...5 sorokat írja ki.
```

Nem lehet a fájl végén túl vagy fordított sorrendben sorokat kiíratni; a **$,$+1p** és **$,1p** jellegű parancsok szabálytalanok.

Az **l** listázóparancs olyan formátumban ír, amelyben az összes karakter látható; ez vezérlőkarakterek keresésére, szóközök és tabulátorok megkülönböztetésére stb. használható.

Minták

Ha egy fájl nem csupán néhány soros, akkor kellemetlen egy adott sor keresése miatt az egészet kiíratni. Az **ed**-del lehet adott mintát tartalmazó sorokat keresni: */minta/* megkeresi a minta következő előfordulását:

```
$ ed vers
232
/bolhák/                Keresi a következő bolhák-at tartalmazó sort.
Nagy bolhák hátán kis bolhák:
/bolhák/                Keresi a következőt.
S még kisebb bolhák megtoldják
//                      Keresi a következő ugyanilyen mintát.
Nagy bolhák rendje még nagyobb
??                      Visszafelé keresi ugyanazt a mintát.
S még kisebb bolhák megtoldják
```

Az **ed** megjegyzi az utoljára alkalmazott mintát, ezért az ismételt keresés a // parancscsal kérhető. Visszafelé a **?minta?** és ?? parancsokkal lehet haladni.

A /.../ és ?...? keresés a szöveg mindkét végénél „körbemegy":

```
$p                      Kiírja az utolsó sort. ( p opcionális.)
jön még nagyobb, s a többi.
/bolhák/                Következő bolhák a legelső.
Nagy bolhák hátán kis bolhák:
??                      Az elején át körbeugrik a végére.
bolhák rendjét előzi;
```

Az olyan minta, mint a */bolhák/*, ugyanolyan sorszám, mint az 1 vagy a $, és ilyen értelemben is használható:

```
1,/bolhák/p             Kiírja az első sortól a következő bolhák-ig.
?bolhák?+1,$p           Kiírja az előző bolhák +1 sortól végig.
```

Hol is vagyunk?

Az **ed** feljegyzi, hogy melyik volt az a sor, amelyben utoljára csináltunk valamit: kiíratást, szöveghozzátételt vagy fájlolvasást. Ennek a sornak pont (.) a neve – ez az *aktuális sor*. Mindegyik parancsnak van hatása a pontra; rendszerint a parancs által érintett utolsó sor lesz a pont. A pontot ugyanúgy használhatjuk, mint a $-t vagy az 1,... számokat:

```
$ ed vers
232
.                       Kiírja az aktuális sort; beolvasás után ugyanaz, mint $.
jön még nagyobb, s a többi.
.-1,.p                  Kiírja az előző és az aktuális sort.
S még nagyobbakét ... tudhatod,
jön még nagyobb, s a többi.
```

A sorszám kifejezések rövidíthetők:

Rövidítés:	*Ugyanaz, mint:*	*Rövidítés:*	*Ugyanazt, mint:*
−	.−1	+	.+1
−− vagy −2	.−2	++ vagy +2	.+2
−n	.−n	+n	.+n
$−	$−1	.3	.+3

Hozzáfűzés, csere, törlés, beszúrás

Az **a** parancs (*append*) a megadott sor után illeszt be sorokat; a **d** parancs (*delete*) sorokat töröl; az **i** parancs (*insert*) a megadott sor elé illeszt sorokat; a **c** parancs (*change*) a sorokat módosító *delete* és *insert* kombinációja.

na	Az **n**-edik sor után illeszti a szöveget.
ni	Az **n**-edik sor elé szúrja be a szöveget.
m,nd	Az **m...n** sorokat törli.
m,nc	Az **m...n** sorokat cseréli.

Ha nincs sorszám, akkor alapértelmezés a pont. A **a**, **c** és **i** parancsoknak szóló új szöveget egy csak egyetlenegy pontból (.) álló sorral kell lezárni; az aktuális sor az utolsó hozzátett sor. Törléskor a pont az utolsó törölt sor utáni sorra áll, de a $ sor utánra nem mehet.

0a	Az elejére illeszt szöveget (ugyanaz, mint **1i**).
dp	Az aktuális sort törli, a következőt (vagy ha **$**-nál van, az utolsót) kiírja.
.,$dp	Innen a végéig töröl, az új utolsót kiírja.
1,$d	Mindent töröl.
?minta?,.−1d	Az előző **minta** sortól a pont előttiig töröl.
$dp	Törli az utolsót, kiírja az új utolsót.
$c	Cseréli az utolsó sort. (**$a** az utolsó utánra illeszt.)
1,$c	Mindent cserél.

Helyettesítés; visszacsinálás

Keserves dolog néhány betű megváltoztatása miatt egy egész sort újragépelni! Az **s** parancs (*substitute*) egy betűláncot egy másikra cserél:

s/régi/új/	Az aktuális sorban az első **régi** helyére **új** kerül.
s/régi/új/p	Az első **régi** helyére **új** kerül és kiírja a sort.
s/régi/új/g	Az aktuális sorban minden **régi** helyére **új** kerül.
s/régi/új/gp	Mindegyik **régi** helyére **új** kerül és kiírja a sort.

Ha nem áll a végén g betű, akkor a sorban csak a minta balról vett első előfordulását cseréli. Az **s** parancs nem írja ki a megváltoztatott sort, ehhez **p** betűt kell a végére írni. Az **ed** parancsok általában kiírás nélkül dolgoznak, de a legtöbb után **p** parancsot írva kinyomtathatjuk az eredményt.

Ha a helyettesítés nem a kívánt módon sikerült, akkor az **u** paranccsal (*undo*) visszacsinálhatjuk a legutolsó helyettesítést. A pontot ehhez a helyettesített sorra kell állítani.

u	A legutolsó helyettesítést visszacsinálja.
up	A legutolsó helyettesítést visszacsinálja és kiírja az eredetit.

Az **s** parancs előtt is állhat egy vagy két, a változtatandó sorokat jelző sorszám (hasonlóan a **p** és **d** parancsokhoz):

/régi/s/régi/új/	Megkeresi a következő **régi**-t és **új**-ra cseréli.
/régi/s//új/	Megkeresi a következő **régi**-t és **új**-ra cseréli (a mintára emlékszik).
1,$s/régi/új/p	Mindegyik sorban az első **régi**-t **új**-ra cseréli és kiírja az utolsó változtatott sort.
1,$s/régi/új/gp	Mindegyik sorban mindegyik **régi**-t **új**-ra cseréli és kiírja az utolsó változtatott sort.

Figyeljük meg, hogy 1,$s minden sorra alkalmazza az **s** parancsot, de mindig csak a balról vett első egyezésre érti; az összes előfordulás cseréjéhez meg kell adni a végén a **g** parancsot! Továbbá a **p** csak az utolsó változtatott sort írja ki; az összes változtatott sor kiírásához a rövidesen tárgyalandó *global* parancsra lesz szükségünk.

Az & jel egy rövidítés; ha az **s** parancsban a jobb oldalon ez a jel megjelenik, akkor helyére az kerül, amivel a bal oldala egyezést mutatott:

s/nagy/nagyon &/	**nagy** helyére **nagyon nagy** kerül.
s/nagy/& &/	**nagy** helyére **nagy nagy** kerül.
s/.*/(&)/	A teljes sor zárójelbe kerül (.* jelentését l. később).
s/és/\&/	**és** helyére **&** kerül (\ kikapcsolja a különleges jelentést).

Metakarakterek és szabályos kifejezések

Ahogyan a burok számára a * és > és | stb. karaktereknek, ugyanúgy az **ed** számára is bizonyos karaktereknek különleges jelentése van, ha azok keresési mintában vagy egy **s** parancs bal oldali részében állnak. Az ilyen karaktereket *metakarakereknek*, az ezeket használó mintákat *szabályos kifejezéseknek* nevezzük. A karaktereket és jelentésüket az F.1. táblázatban soroljuk fel; az alábbi példák a táblázattal együtt olvasandók. Bármely karakter különleges jelentését kikapcsolja az elé írt fordított törtvonal.

Minta:	*Egyezések:*
/^$/	Üres sor, azaz csak újsor-karakter.
/./	Nemüres, azaz legalább egy karakter.
/^/	Minden sor.
/dolog/	**dolog** a sorban bárhol.
/^dolog/	**dolog** a sor elején.
/dolog$/	**dolog** a sor végén.
/^dolog$/	Olyan sor, amelyben csak **dolog** van.

A szerkesztőprogram szabályos kifejezései

c	bármely nem speciális c karakter saját magával egyezik
\c	a c karakter speciális jelentését kikapcsolja
^	sor elején egyezés
$	sor végén egyezés
.	egyetlen karakterrel egyezés
[...]	bármely ... közüli egyetlen karakter; az a – z tartomány megengedett forma
[^...]	bármely egyetlen karakter, amely nincs a ... között; tartományok megengedettek
r*	r nulla vagy több előfordulása, ahol r egy karakter, . vagy [...]
&	s jobb oldalán az egyezett minta ismétlése
\(...\)	jelzett szabályos kifejezés; az egyező láncra mind a bal, mind a jobb oldalon \1 stb.-vel hivatkozhatunk
	Az újsorral semmilyen szabályos kifejezés sem egyezik

/dolog.$/	**dolog** és bármely karakter a sor végén.
/dolog\.$/	**dolog.** a sor végén.
/\/dolog\//	**/dolog/** a sorban bárhol.
/[dD]olog/	**dolog** vagy **Dolog** a sorban bárhol.
/dolog[0–9]/	**dolog** és utána egy számjegy.
/dolog[^0–9]/	**dolog** és utána nem számjegy.
/dolog[0–9][^0–9]/	**dolog** és utána számjegy, nem számjegy.
/dolog1.*dolog2/	**dolog1** és utána bármilyen lánc, majd **dolog2** is.
/^dolog1.*dolog2$/	**dolog1** az elején és **dolog2** a végén.

A * karaktert tartalmazó szabályos kifejezések a balról vett első egyezést választják és olyan hosszan veszik, ahogyan csak lehet. Megjegyezzük, hogy x* nulla karakterrel, xx* egy vagy több karakterrel mutathat egyezést.

A *global* parancsok

A *global* parancsok, a g és a v, egy vagy több parancsot egy szabályos kifejezés által meghatározott sorhalmazra alkalmaznak. A g parancsot leggyakrabban egy sorhalmaz kiíratására, helyettesítésére, törlésére használjuk:

m,ng/hely/parancs Minden m...n közötti, **hely**-lyel egyező sorra végrehajtja a **parancsot**-ot.
m,nv/hely/parancs Minden m...n közötti, **hely**-lyel nem egyező sorra végrehajtja a **parancs**-ot.

A **g** és **v** parancs működése az eléjük írt sorszámokkal határolható; az alapértelmezés szerinti tartomány az 1,$.

g/.../p Kiírja a ... szabályos kifejezéssel egyező összes sort.
g/.../d Törli a ... szabályos kifejezéssel egyező összes sort.
g/.../s//*csere*/p Minden sor első ...-ja helyére **csere**-t tesz és kiírja a megváltoztatott sorokat.
g/.../s//*csere*/gp Minden ... helyére **csere**-t tesz és kiírja a megváltoztatott sorokat.
g/.../s/*minta*/*csere*/ A ...-tal egyező sorokban az első **minta** helyére **csere**-t tesz.
g/.../s/*minta*/*csere*/p A ...-tal egyező sorokban az első **minta** helyére **csere**-t tesz és kiírja a megváltoztatott sorokat.
g/.../s/*minta*/*csere*/gp A ...-tal egyező sorokban minden **minta** helyére **csere**-t tesz és kiírja a megváltoztatott sorokat.
v/.../s/*minta*/*csere*/gp A ...-tal nem egyező sorokban minden **minta** helyére **csere**-t tesz és kiírja a megváltoztatott sorokat.
v/^$/p Kiír minden nemüres sort.
g/.../*parancs1*\ Egyetlen **g** több parancsra vonatkozik, ha az utolsó kivételével mindegyik után \ jelet teszünk.
parancs2\
parancs3

A **g** és **v** parancsok által vezérelt parancsokban is lehetnek sorszámok. A pont rendre a kiválasztott sorokra állítódik.

g/*dolog*/.,.+1p Kiír minden **dolog** tartalmú sort és az aktuális sort követőt.
g/^\.EQ/.1,/^\.EN/−s/alfa/béta/gp
.EQ és **.EN** között minden **alfa** helyére **béta**-t tesz és kiírja a megváltoztatott sorokat.

Sorok átmozgatása és másolása

Az **m** parancs a sorok egy folytonos csoportját átmozgatja; a **t** parancs a sorok egy csoportjáról készít másolatot.

*m,n*m*d* Az m...n sorokat átmozgatja a **d** sor utánra.
*m,n*t*d* Az m...n sorokról másolatot helyez el a **d** sor után.

Ha nem adunk meg kezdősort, a parancsok az aktuális sort feltételezik. A d célsor nem lehet m és n − 1 közötti. Az **m** és **t** parancsok néhány gyakran használt formája:

m+ Az aktuális sort a következő utánra mozgatja (kölcsönös csere).
m−2 Az aktuális sort az előző sor elé mozgatja.
m−− Ugyanaz: −− ugyanaz, mint −**2**.
m− Semmit sem csinál.

m$	Az aktuális sort a végére (**m0** az elejére) teszi.
t.	Az aktuális sort megkettőzi (**t$** a végén kettőz).
−,.t.	Az előző és az aktuális sort megkettőzi.
1,t	A teljes sorhalmazt megkettőzi.
g/^/m0	Megfordítja a sorok sorrendjét.

.

Megjelölés és sorszámok

Az = parancs kiírja a $ sor sorszámát (ostoba alapértelmezés), a .= kiírja az aktuális sor sorszámát stb. A pont változatlan marad.

A **kc** parancs a megcímzett sort a c kisbetűvel *megjelöli;* a sorra ezután **'c** sorszámmal hivatkozhatunk. A **k** parancs nem változtatja meg a pontot. A sorok ilyen megjelölése nagy szövegrészek mozgatására használható, ugyanis állandóan a sorhoz hozzárendelve marad, mint az alábbi parancssorozatban:

/.../ka	Megkeresi a ... sort és **a**-val megjelöli.
/.../kb	Megkeresi a ... sort és **b**-vel megjelöli.
'a,'bp	Ellenőrzésként kiírja.
/.../	Megkeresi a célsort.
'a,'bm.	A kiválasztott sorokat a célsor utánra átmozgatja.

Sorok egybefűzése, tördelése, átrendezése

A **j** paranccsal sorokat fűzhetünk egybe (szóköz beiktatása nélkül):

m,nj	Az m...n sorokat egybefűzi egy sorrá.

Az alapértelmezés szerinti tartomány a **.,.+1,** úgyhogy:

jp	Az aktuális sort a következővel egybefűzi és kiírja.
−,.jp	Az előző sort az aktuálissal egybefűzi és kiírja.

Az **s** helyettesítő paranccsal újsort beszúrva a sorokat tördelhetjük:

s/egyikmásik/egyik\	A sort két részre tördeli.
másik/	...
s/ /\	A sort minden szóköznél tördeli:
/g	egy sor egy szó!

A pont az utolsó létrehozott sornál marad.

Hogy az egyezést adó szabályos kifejezések részeiről is beszélhessünk, ne csak az egészről, bevezetjük a *jelzett szabályos kifejezéseket:* ha egy szabályos kifejezésben a \(...\) szerkezet szerepel, akkor az egésznek arra a részére, amely ezzel egyezik, mind a bal oldalon, mind a jobb oldalon, a **\1** jelöléssel hivatkozhatunk. Legfeljebb kilenc jelzett kifejezés lehet, amelyeket **\1**, **\2** stb. jelölhet.

s/\(...\)\(.*\)/\2\1/	Az első három karaktert a végére teszi.
/\(..*\)\1/	Ismétlődő csatlakozó láncot tartalmazó sorokat keres.

Fájlkezelő parancsok

Az **r** és **w** író- és olvasóparancsok (*read* és *write*) előtt sorszámok állhatnak:

nr fájl	Fájlolvasás, az olvasott részt az n-edik sor után helyezi el; a pontot az utolsó olvasott sorra állítja.
m,nw fájl	Az m...n sorokat fájlba írja; a pont változatlan.
m,nW fájl	Az m...n sorokat a fájl végéhez írja; a pont változatlan.

A **w** és **W** parancsok alapértelmezése a teljes fájl. Az **r** parancs **n** paraméterének alapértelmezése a $, ami ismételten szerencsétlen választás. Óvakodjunk!

Az **ed** megjegyzi a – parancssorból vagy **r** vagy **w** parancsból származó – először használt fájlnevet. Az **f** parancs kiírja vagy megváltoztatja a megjegyzett nevet:

f	Kiírja a megjegyzett fájl nevét.
f *fájl*	A megjegyzett fájl nevét **fájl**-ra állítja.

Az **e** parancs (*edit*) a megjegyzett fájllal vagy egy új fájllal újraindítja az **ed**-et:

e	Szerkesztés kezdete a megjegyzett fájlra.
e *fájl*	Szerkesztés kezdete a **fájl**-ra.

Az **e** parancs ugyanolyan védelmet biztosít, mint a **q** parancs: ha nem írtuk ki a változtatásokat, először figyelmeztet erre. Az **E** parancs figyelmeztetés nélkül újraindítja az **ed**-et. Némely rendszerben az **ed** az **e**-hez van láncolva, úgyhogy a szerkesztőn belülről és kívülről ugyanazzal az **e** *fájlnév* paranccsal indítható.

Titkosítás

A fájlok az **x** parancs segítségével íráskor titkosíthatók, illetve olvasásra engedélyezhetők; a parancs jelszót kérdez. A titkosítási eljárás ugyanaz, mint **crypt(1)**. Némely rendszerben az **x** parancs helyett **X** van (nagybetűvel), hogy jobban elkerülhető legyen a nem szándékolt titkosítás.

A parancsok összefoglalása

Az F.2. táblázat az **ed** parancsok, az F.3. táblázat az érvényes sorszámok összefoglalása. Mindegyik parancs előtt nulla, egy vagy két sorszám állhat jelezve, hogy mely sorokra, ill., hogy az alapértelmezés szerintiekre vonatkozik a parancs. A legtöbb parancs után p-vel (vagy *list* formában l-lel) ki lehet íratni az utolsó érintett sort.

Az **ed** parancsok összefoglalása

.a	szöveg hozzátétele csak . tartalmú sor begépeléséig
.,.c	sorok cseréje; új szöveg vége, mint **a**-nál
.,.d	sorok törlése
e *fájl*	újraindítás *fájl*-ra. **E** akkor is újraindít, ha a változtatások nincsenek kiírva
f *fájl*	a megjegyzett fájl nevét *fájl*-ra állítja
1,$g/*szk***/***parancsok*	az *szk* szabályos kifejezéssel egyezést mutató sorok mindegyikére végrehajtja az **ed** *parancsok*-at; az egyes parancsokat \újsor-ral kell elválasztani
.i	szövegbeszúrás a sor elé; vége mint **a**-nál
.,.+1j	sorok egybefűzése
.k*c*	sor megcímkézése a *c* betűvel
.,.l	sorok listázása, a láthatatlan karakterek láthatóvá tételével
.,.m*sor*	sorok átmozgatása sor utánra
.,.p	sorok kiíratása
q	kilépés. A **Q** akkor is kilép, ha a változtatások nincsenek kiírva
$r *fájl*	fájl olvasása
.,.s/*szk***/***új***/**	*szk*-val egyező helyettesítése *új*-jal
.,.t*sor*	sorokat a *sor* utánra másol
.u	az utolsó változtatott sort (csak egyet) visszacsinálja
1,$v/*szk***/***parancsok*	az *szk*-val nem egyező sorok mindegyikére végrehajtja az **ed** *parancsok*-at
1,$w *fájl*	a sorokat *fájl*-ba írja; **W** felülírás helyett hozzáfűzi
x	titkosító mód (vagy **ed −x** *fájlnév*)
$=	sorszám kiírása
!*parancssor*	*parancssor* UNIX parancs végrehajtása
(.+1)*újsor*	sor kiírása

A pont általában az utolsó érintett sor; az **f**, **k**, **w**, **x**, **=** és **!** parancsok azonban nem változtatják.

Feladat. Ha úgy gondoljuk, mindent tudunk az **ed**-ről, próbálkozzunk meg a szerkesztőteszttel; l. **quiz(6)**. □

Az **ed** sorszámformátumok összefoglalása

n	abszolút sorszám, $n = 0, 1, 2, \ldots$
.	aktuális sor
$	a szöveg utolsó sora
/szk/	a következő, *szk*-val egyezést mutató sor; $ és 1 körül körbemegy
?szk?	az előző, *szk*-val egyezést mutató sor; 1 és $ között körbemegy
'c	a *C*-vel megcímkézett sor
N1 ± *n*	az N1 ± *n* sor (additív kombináció)
N1,N2	az N1...N2 sorok
N1;N2	a pontot N1-re állítja, majd kiszámítja N2-t
	N1 és N2 a fentiek bármelyike lehet

2. függelék

A hoc kézikönyv

Hoc – Interaktív lebegőpontos aritmetikai nyelv

Brian Kernighan
Rob Pike

KIVONAT

A *hoc* lebegőpontos kifejezések egyszerű programozható értelmezője. C nyelvhez hasonló vezérlés-ellenőrzése, függvénydefiníciója van, és rendelkezik a szokásos beépített numerikus függvényekkel, mint pl. a koszinusz és a logaritmus.

1. Kifejezések

A *hoc* a C nyelvhez hasonlóan egy kifejezésnyelv; van ugyan számos vezérlés-ellenőrző utasítása, de a legtöbb utasítása (mint pl. az értékadások) olyan kifejezés, amelynek értéke közömbös. Például az = értékadó operátor a jobb oldali operandus értékét hozzárendeli a bal oldali operandushoz, tehát értéket közvetít. A többszörös értékadás is működik. A kifejezés nyelvtana a következő:

kifejezés: *szám*
/ *változó*
/ (*kifejezés*)
/ *kifejezés binop kifejezés*
/ *unop kifejezés*
/ *függvény* (*argumentumok*)

A számok lebegőpontosak. A bemenet a **scanf(3)** által felismerhető formátumú: számjegyek, tizedespont, számjegyek, *e* vagy *E*, előjeles kitevő. Legalább egy számjegy vagy tizedespont kötelező; a többi összetevő opcionális.

A változónevek alakja egy betű, amelyet betűkből és számokból álló lánc követ. A **binop** kétváltozós operátort jelent, ami lehet például összeadás vagy logikai összehasonlítás; a **unop** jelöli a két tagadás operátort, a ! (logikai tagadás, logikai nem) vagy – (aritmetikai tagadás, előjelváltás) egyike lehet. Az operátorokat az F.4. táblázat tartalmazza.

F.4. táblázat

Műveleti jelek csökkenő precedenciarendben

^	hatványozás (FORTRAN-ban**), jobbról asszociatív
! –	(egyváltozós) logikai és aritmetikai tagadás
* /	szorzás, osztás
+ –	összeadás, kivonás
> >=	relációs jelek: nagyobb, nagyobb egyenlő
< <=	kisebb, kisebb egyenlő
=! =	egyenlő, nem egyenlő (mind azonprecedenciával)
&&	logikai ÉS (mindkét operandust mindig kiszámítja)
||	logikai VAGY (mindkét operandust mindig kiszámítja
=	értékadás (jobbról asszociatív)

F.5. táblázat

Beépített függvények

abs(x)	$\mid x \mid$, x abszolút értéke
atan(x)	x arkusz tangense
cos(x)	x koszinusza
exp(x)	e^x hatványfüggvény
int(x)	x nullához közelebbi egészrésze
log(x)	$\log_e(x)$
log10(x)	x tízes alapú logartimusa
sin(x)	x szinusza
sqrt(x)	x négyzetgyöke

A függvények, mint látni fogjuk, a felhasználó által definiálhatók. A függvényargumentumok vesszőkkel elválasztott kifejezések. A beépített függvényeket, amelyek mind egyargumentumosak, az F.5. táblázatban soroljuk fel.

A logikai kifejezések értékei az 1,0 (igaz) és a 0,0 (hamis). Miként a C nyelvben, itt is minden nemnulla érték igazat jelent. Ahogyan a lebegőpontos számoknál általában, az egyenlőség szerinti összehasonlítások eredményei eleve gyanúsak.

A *hoc* néhány beépített állandóval is rendelkezik, amelyek az F.6. táblázatban láthatók.

Beépített állandók

DEG	57,295779513082320876 80	$180/\pi$, fok/radián
E	2,718281828459045235 36	e, természetes logaritmus alapszáma
GAMMA	0,577215664901532860 60	γ, Euler – Mascheroni-állandó
PHI	1,618033988749894848 20	$(\sqrt{5}+1)/2$, aranymetszés aránya
PI	3,141592653589793238 46	π, végtelen nemperiodikus tizedestört

2. Utasítások és vezérlésátadás

A *hoc* utasítások nyelvtana a következő:

utasítás: *kifejezés*
/ *változó = kifejezés*
/ *eljárás (argumentumlista)*
/ *while (kifejezés) utasítás*
/ *while (kifejezés) utasítás else utasítás*
/ *{ utasításlista }*
/ *print kifejezéslista*
/ *return opcionális kifejezés*

utasításlista: *(semmi)*
/ *utasításlista utasítás*

Az értékadás alapértelmezésben utasítás, nem pedig kifejezés, így az interaktív üzemmódban begépelt értékadások nem írják ki értéküket.

Megjegyezzük, hogy a *hoc* nyelvben nincs pontosvessző: az utasításokat újsor zárja. Ez kissé különös dolgokat okozhat. Az alábbiak szabályos **if** utasítások:

```
if (x < 0) print(y) else print(z)

if (x < 0) {
    print(y)
} else {
    print(z)
}
```

A második **if** utasításban a kapcsos zárójelek nem hagyhatók el: ha elhagynánk, akkor az **if** utáni újsor-karakter lezárná az utasítást, és ez szintaktikus hibát okozna.

A *hoc* vezérlésátadó lehetőségeinek szintaktikája és szemantikája lényegében a C nyelvével azonos, de nincs **break** és **continue** utasítás.

3. Bemenet és kimenet: read és print

A **read** bemeneti függvénynek, mint a többi beépített függvénynek, egy argumentuma van. A beépített függvényekétől eltérően az argumentum nem kifejezés, hanem változónév. A következő (az előbbi definíció szerinti) számot a szabványos bemenetről beolvassa és a megnevezett változóhoz hozzárendeli. A **read** által visszaadott érték 1 (igaz), ha egy értéket beolvasott, és 0 (hamis), ha fájlvéget vagy hibát talált.

A kimenetet a **print** utasítás állítja elő. A **print** argumentumai kifejezések és idézőjelek közötti láncok vesszőkkel elválasztott listái. Az új sorokat meg kell adni, a **print** automatikusan sosem írja ki azokat.

Megjegyezzük, hogy amíg a **read** egy különleges beépített függvény és egyetlen zárójelezett argumentuma van, addig a **print** utasításhoz vesszőkkel elválasztott, nem zárójelezett lista tartozik:

```
while (read(x)) {
        print "az érték ", x, "\n"
}
```

4. Függvények és eljárások

A *hoc* nyelvben a függvények és eljárások nem azonosak, bár definíciójuk megegyezik. A különbség a futás közbeni hibaellenőrzésben van: hiba esetén az eljárás visszaad egy értéket, a függvény pedig *nem*.

A definíciók szintaxisa a következő:

```
függvény:    függvénynév() utasítás

eljárás:     eljárásnév() utasítás
```

ahol a *név* a beépített függvényeké kivételével bármely változónév lehet. A definíció a nyitó kapcsos zárójelig vagy utasításig egy sorban kell, hogy legyen ugyanúgy, mint előbb az **if** utasítás esetében.

A C nyelvtől eltérően a függvénytest vagy eljárástest bármilyen utasítás lehet, nem csupán összetett (kapcsos zárójelek közé zárt) utasítás. Mivel a *hoc* nyelvben a pontosvesszőnek nincs jelentése, az üres eljárástest üres kapcsos zárójelpárból áll.

A függvények és eljárások meghívásukkor vesszőkkel elválasztott argumentumokat vehetnek át. Az argumentumokra ugyanúgy lehet hivatkozni, mint a buroknál: $3 a harmadik argumentumot jelenti. Az argumentumok érték szerint kerülnek átadásra, és a függvényeken belül szemantikusan azonosak a változókkal. Hibát okoz, ha nagyobb sorszámú argumentumra hivatkozunk, mint amennyit a rutinnak átadtunk. Jóllehet, ezt futás közben ellenőrzi a rendszer, mégis lehet egy rutinnak változó számú argumentuma, ha kezdeti argumentumai befolyásolják az argumentumok számát (mint a C-ben a **printf** függvénynél).

A függvények és eljárások lehetnek rekurzívok, de a verem mélysége korlátozott (kb. száz hívásra). Íme az Ackermann-függvény *hoc* definíciója:

```
$ hoc
func ack() {
        if ($1 == 0) return $2+1
        if ($2 == 0) return ack($1-1, 1)
        return ack($1-1, ack($1, $2-1))
}
ack(3, 2)
    29
ack(3, 3)
    61
ack(3, 4)
hoc: stack too deep near line 8
...
```

5. Példák

A Stirling-formula:

$$n! \sim \sqrt{2n\pi}(n/e)^n(1+\frac{1}{12n})$$

```
$ hoc
func stirl() {
    return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
}
stirl(10)
    3628684.7
stirl(20)
    2.4328818e+18
```

Az **n!** faktoriális függvény:

```
func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
```

A faktoriálisok és a Stirling-közelítés hányadosa:

```
i = 9
while ((i = i+1) <= 20) {
      print i, " ", fac(i)/stirl(i), "\n"
}
10   1.0000318
11   1.0000265
12   1.0000224
13   1.0000192
14   1.0000166
15   1.0000146
16   1.0000128
17   1.0000114
18   1.0000102
19   1.0000092
20   1.0000083
```

A hoc lista*

```
*****   hoc.y   ***************************************************************

%{
#include "hoc.h"
#define code2(c1,c2)     code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
        Symbol  *sym;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
        int     narg;   /* number of arguments */
}
%token  <sym>   NUMBER STRING PRINT VAR BLTIN UNDEF WHILE IF ELSE
%token  <sym>   FUNCTION PROCEDURE RETURN FUNC PROC READ
%token  <narg>  ARG
%type   <inst>  expr stmt asgn prlist stmtlist
%type   <inst>  cond while if begin end
%type   <sym>   procname
%type   <narg>  arglist
%right  '='
%left   OR
%left   AND
%left   GT GE LT LE EQ NE
%left   '+' '-'
%left   '*' '/'
%left   UNARYMINUS NOT
%right  '^'
%%
list:     /* nothing */
        | list '\n'
        | list defn '\n'
        | list asgn '\n'  { code2(pop, STOP); return 1; }
        | list stmt '\n'  { code(STOP); return 1; }
        | list expr '\n'  { code2(print, STOP); return 1; }
        | list error '\n' { yyerrok; }
        ;
asgn:     VAR '=' expr { code3(varpush,(Inst)$1,assign); $$=$3; }
        | ARG '=' expr
          { defnonly("$"); code2(argassign,(Inst)$1); $$=$3;}
        ;
stmt:     expr  { code(pop); }
        | RETURN { defnonly("return"); code(procret); }
        | RETURN expr
                { defnonly("return"); $$=$2; code(funcret); }
        | PROCEDURE begin '(' arglist ')'
                { $$ = $2; code3(call, (Inst)$1, (Inst)$4); }
        | PRINT prlist  { $$ = $2; }
        | while cond stmt end {
                ($1)[1] = (Inst)$3;     /* body of loop */
                ($1)[2] = (Inst)$4; }   /* end, if cond fails */
        | if cond stmt end {    /* else-less if */
                ($1)[1] = (Inst)$3;     /* thenpart */
                ($1)[3] = (Inst)$4; }   /* end, if cond fails */
```

* Az eredeti könyvben levő listát változtatás nélkül közöljük. A szövegkörnyezetben viszont a megjegyzéseket magyarul adtuk meg. *(A szerk.)*

```
        | if cond stmt end ELSE stmt end {      /* if with else */
              ($1)[1] = (Inst)$3;     /* thenpart */
              ($1)[2] = (Inst)$6;     /* elsepart */
              ($1)[3] = (Inst)$7; }   /* end, if cond fails */
        | '{' stmtlist '}'      { $$ = $2; }
        ;
cond:     '(' expr ')'  { code(STOP); $$ = $2; }
        ;
while:    WHILE { $$ = code3(whilecode,STOP,STOP); }
        ;
if:       IF    { $$ = code(ifcode); code3(STOP,STOP,STOP); }
        ;
begin:    /* nothing */          { $$ = progp; }
        ;
end:      /* nothing */          { code(STOP); $$ = progp; }
        ;
stmtlist: /* nothing */          { $$ = progp; }
        | stmtlist '\n'
        | stmtlist stmt
        ;
expr:     NUMBER { $$ = code2(constpush, (Inst)$1); }
        | VAR    { $$ = code3(varpush, (Inst)$1, eval); }
        | ARG    { defnonly("$"); $$ = code2(arg, (Inst)$1); }
        | asgn
        | FUNCTION begin '(' arglist ')'
              { $$ = $2; code3(call,(Inst)$1,(Inst)$4); }
        | READ '(' VAR ')' { $$ = code2(varread, (Inst)$3); }
        | BLTIN '(' expr ')' { $$=$3; code2(bltin, (Inst)$1->u.ptr); }
        | '(' expr ')'  { $$ = $2; }
        | expr '+' expr { code(add); }
        | expr '-' expr { code(sub); }
        | expr '*' expr { code(mul); }
        | expr '/' expr { code(div); }
        | expr '^' expr { code (power); }
        | '-' expr   %prec UNARYMINUS   { $$=$2; code(negate); }
        | expr GT expr  { code(gt); }
        | expr GE expr  { code(ge); }
        | expr LT expr  { code(lt); }
        | expr LE expr  { code(le); }
        | expr EQ expr  { code(eq); }
        | expr NE expr  { code(ne); }
        | expr AND expr { code(and); }
        | expr OR expr  { code(or); }
        | NOT expr      { $$ = $2; code(not); }
        ;
prlist:   expr                  { code(prexpr); }
        | STRING                { $$ = code2(prstr, (Inst)$1); }
        | prlist ',' expr       { code(prexpr); }
        | prlist ',' STRING     { code2(prstr, (Inst)$3); }
        ;
defn:     FUNC procname { $2->type=FUNCTION; indef=1; }
            '(' ')' stmt { code(procret); define($2); indef=0; }
        | PROC procname { $2->type=PROCEDURE; indef=1; }
            '(' ')' stmt { code(procret); define($2); indef=0; }
        ;
procname: VAR
        | FUNCTION
        | PROCEDURE
        ;
arglist:  /* nothing */          { $$ = 0; }
        | expr                  { $$ = 1; }
        | arglist ',' expr      { $$ = $1 + 1; }
        ;
%%


        /* end of grammar */
#include <stdio.h>
#include <ctype.h>
char    *progname;
int     lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int     indef;
char    *infile;        /* input file name */
FILE    *fin;           /* input file pointer */
char    **gargv;        /* global argument list */
int     gargc;
```

```
int c;  /* global for use by warning() */
yylex()         /* hoc6 */
{
        while ((c=getc(fin)) == ' ' || c == '\t')
                ;
        if (c == EOF)
                return 0;
        if (c == '.' || isdigit(c)) {   /* number */
                double d;
                ungetc(c, fin);
                fscanf(fin, "%lf", &d);
                yylval.sym = install("", NUMBER, d);
                return NUMBER;
        }
        if (isalpha(c)) {
                Symbol *s;
                char sbuf[100], *p = sbuf;
                do {
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("name too long", sbuf);
                        }
                        *p++ = c;
                } while ((c=getc(fin)) != EOF && isalnum(c));
                ungetc(c, fin);
                *p = '\0';
                if ((s=lookup(sbuf)) == 0)
                        s = install(sbuf, UNDEF, 0.0);
                yylval.sym = s;
                return s->type == UNDEF ? VAR : s->type;
        }
        if (c == '$') { /* argument? */
                int n = 0;
                while (isdigit(c=getc(fin)))
                        n = 10 * n + c - '0';
                ungetc(c, fin);
                if (n == 0)
                        execerror("strange $...", (char *)0);
                yylval.narg = n;
                return ARG;
        }
        if (c == '"') { /* quoted string */
                char sbuf[100], *p, *emalloc();
                for (p = sbuf; (c=getc(fin)) != '"'; p++) {
                        if (c == '\n' || c == EOF)
                                execerror("missing quote", "");
                        if (p >= sbuf + sizeof(sbuf) - 1) {
                                *p = '\0';
                                execerror("string too long", sbuf);
                        }
                        *p = backslash(c);
                }
                *p = 0;
                yylval.sym = (Symbol *)emalloc(strlen(sbuf)+1);
                strcpy(yylval.sym, sbuf);
                return STRING;
        }
        switch (c) {
        case '>':       return follow('=', GE, GT);
        case '<':       return follow('=', LE, LT);
        case '=':       return follow('=', EQ, '=');
        case '!':       return follow('=', NE, NOT);
        case '|':       return follow('|', OR, '|');
        case '&':       return follow('&', AND, '&');
        case '\n':      lineno++; return '\n';
        default:        return c;
        }


}
backslash(c)    /* get next char with \'s interpreted */
        int c;
{
        char *index();  /* 'strchr()' in some systems */
        static char transtab[] = "b\bf\fn\nr\rt\t";
        if (c != '\\')
                return c;
```

```
        c = getc(fin);
        if (islower(c) && index(transtab, c))
                return index(transtab, c)[1];
        return c;
}

follow(expect, ifyes, ifno)     /* look ahead for >=, etc. */
{
        int c = getc(fin);

        if (c == expect)
                return ifyes;
        ungetc(c, fin);
        return ifno;
}

defnonly(s)     /* warn if illegal definition */
        char *s;
{
        if (!indef)
                execerror(s, "used outside definition");
}

yyerror(s)      /* report compile-time error */
        char *s;
{
        warning(s, (char *)0);
}

execerror(s, t) /* recover from run-time error */
        char *s, *t;
{
        warning(s, t);
        fseek(fin, 0L, 2);              /* flush rest of file */
        longjmp(begin, 0);
}

fpecatch()      /* catch floating point exceptions */
{
        execerror("floating point exception", (char *) 0);
}

main(argc, argv)        /* hoc6 */
        char *argv[];
{
        int i, fpecatch();

        progname = argv[0];
        if (argc == 1) {        /* fake an argument list */
                static char *stdinonly[] = { "-" };

                gargv = stdinonly;
                gargc = 1;
        } else {
                gargv = argv+1;
                gargc = argc-1;
        }
        init();
        while (moreinput())
                run();
        return 0;
}


moreinput()
{
        if (gargc-- <= 0)
                return 0;
        if (fin && fin != stdin)
                fclose(fin);
        infile = *gargv++;
        lineno = 1;
        if (strcmp(infile, "-") == 0) {
                fin = stdin;
                infile = 0;
        } else if ((fin=fopen(infile, "r")) == NULL) {
                fprintf(stderr, "%s: can't open %s\n", progname, infile);
                return moreinput();
        }
        return 1;
}
```

```
run()   /* execute until EOF */
{
        setjmp(begin);
        signal(SIGFPE, fpecatch);
        for (initcode(); yyparse(); initcode())
                execute(progbase);
}

warning(s, t)   /* print warning message */
        char *s, *t;
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t)
                fprintf(stderr, " %s", t);
        if (infile)
                fprintf(stderr, " in %s", infile);
        fprintf(stderr, " near line %d\n", lineno);
        while (c != '\n' && c != EOF)
                c = getc(fin);  /* flush rest of input line */
        if (c == '\n')
                lineno++;
}
```

```
*****   hoc.h   ************************************************************
```

```
typedef struct Symbol { /* symbol table entry */
        char    *name;
        short   type;
        union {
                double  val;            /* VAR */
                double  (*ptr)();       /* BLTIN */
                int     (*defn)();      /* FUNCTION, PROCEDURE */
                char    *str;           /* STRING */
        } u;
        struct Symbol   *next;  /* to link to another */
} Symbol;
Symbol  *install(), *lookup();

typedef union Datum {   /* interpreter stack type */
        double  val;
        Symbol  *sym;
} Datum;

extern  Datum pop();
extern  eval(), add(), sub(), mul(), div(), negate(), power();

typedef int (*Inst)();
#define STOP    (Inst) 0

extern  Inst *progp, *progbase, prog[], *code();
extern  assign(), bltin(), varpush(), constpush(), print(), varread();
extern  prexpr(), prstr();
extern  gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();
extern  ifcode(), whilecode(), call(), arg(), argassign();
extern  funcret(), procret();
```

```
*****   symbol.c   *********************************************************
```

```
#include "hoc.h"
#include "y.tab.h"

static Symbol *symlist = 0;  /* symbol table: linked list */

Symbol *lookup(s)       /* find s in symbol table */
        char *s;
{
        Symbol *sp;

        for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;
        return 0;       /* 0 ==> not found */
}

Symbol *install(s, t, d)  /* install s in symbol table */
        char *s;
        int t;
        double d;
{
        Symbol *sp;
        char *emalloc();
```

```
        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist; /* put at front of list */
        symlist = sp;
        return sp;
}

char *emalloc(n)        /* check return from malloc */
        unsigned n;
{
        char *p, *malloc();

        p = malloc(n);
        if (p == 0)
                execerror("out of memory", (char *) 0);
        return p;
}
```

***** code.c **********

```
#include "hoc.h"
#include "y.tab.h"
#include <stdio.h>

#define NSTACK  256

static Datum stack[NSTACK];     /* the stack */
static Datum *stackp;           /* next free spot on stack */

#define NPROG   2000
Inst    prog[NPROG];    /* the machine */
Inst    *progp;         /* next free spot for code generation */
Inst    *pc;            /* program counter during execution */
Inst    *progbase = prog; /* start of current subprogram */
int     returning;      /* 1 if return stmt seen */

typedef struct Frame {  /* proc/func call stack frame */
        Symbol  *sp;    /* symbol table entry */
        Inst    *retpc; /* where to resume after return */
        Datum   *argn;  /* n-th argument on stack */
        int     nargs;  /* number of arguments */
} Frame;
#define NFRAME  100
Frame   frame[NFRAME];
Frame   *fp;            /* frame pointer */

initcode() {
        progp = progbase;
        stackp = stack;
        fp = frame;
        returning = 0;
}

push(d)
        Datum d;
{
        if (stackp >= &stack[NSTACK])
                execerror("stack too deep", (char *)0);
        *stackp++ = d;
}

Datum pop()
{
        if (stackp == stack)
                execerror("stack underflow", (char *)0);
        return *--stackp;
}

constpush()
{
        Datum d;
        d.val = ((Symbol *)*pc++)->u.val;
        push(d);
}

varpush()
{
        Datum d;
        d.sym = (Symbol *)(*pc++);
        push(d);
}
```

```
whilecode()
{
        Datum d;
        Inst *savepc = pc;

        execute(savepc+2);      /* condition */
        d = pop();
        while (d.val) {
                execute(*((Inst **)(savepc)));  /* body */
                if (returning)
                        break;
                execute(savepc+2);      /* condition */
                d = pop();
        }
        if (!returning)
                pc = *((Inst **)(savepc+1)); /* next stmt */
}

ifcode()
{
        Datum d;
        Inst *savepc = pc;      /* then part */

        execute(savepc+3);      /* condition */
        d = pop();
        if (d.val)
                execute(*((Inst **)(savepc)));
        else if (*((Inst **)(savepc+1))) /* else part? */
                execute(*((Inst **)(savepc+1)));
        if (!returning)
                pc = *((Inst **)(savepc+2)); /* next stmt */
}

define(sp)      /* put func/proc in symbol table */
        Symbol *sp;
{
        sp->u.defn = (Inst)progbase;    /* start of code */
        progbase = progp;       /* next code starts here */
}


call()          /* call a function */
{
        Symbol *sp = (Symbol *)pc[0]; /* symbol table entry */
                                      /* for function */
        if (fp++ >= &frame[NFRAME-1])
                execerror(sp->name, "call nested too deeply");
        fp->sp = sp;
        fp->nargs = (int)pc[1];
        fp->retpc = pc + 2;
        fp->argn = stackp - 1;  /* last argument */
        execute(sp->u.defn);
        returning = 0;
}

ret()           /* common return from func or proc */
{
        int i;
        for (i = 0; i < fp->nargs; i++)
                pop();  /* pop arguments */
        pc = (Inst *)fp->retpc;
        --fp;
        returning = 1;
}

funcret()       /* return from a function */
{
        Datum d;
        if (fp->sp->type == PROCEDURE)
                execerror(fp->sp->name, "(proc) returns value");
        d = pop();      /* preserve function return value */
        ret();
        push(d);
}

procret()       /* return from a procedure */
{
        if (fp->sp->type == FUNCTION)
                execerror(fp->sp->name,
                        "(func) returns no value");
        ret();
}
```

```
double *getarg()        /* return pointer to argument */
{
        int nargs = (int) *pc++;
        if (nargs > fp->nargs)
            execerror(fp->sp->name, "not enough arguments");
        return &fp->argn[nargs - fp->nargs].val;
}

arg()   /* push argument onto stack */
{
        Datum d;
        d.val = *getarg();
        push(d);
}

argassign()     /* store top of stack in argument */
{
        Datum d;
        d = pop();
        push(d);        /* leave value on stack */
        *getarg() = d.val;
}

bltin()
{
        Datum d;
        d = pop();
        d.val = (*(double (*)())*pc++)(d.val);
        push(d);
}

eval()          /* evaluate variable on stack */
{
        Datum d;
        d = pop();
        if (d.sym->type != VAR && d.sym->type != UNDEF)
                execerror("attempt to evaluate non-variable", d.sym->name);
        if (d.sym->type == UNDEF)
                execerror("undefined variable", d.sym->name);
        d.val = d.sym->u.val;
        push(d);
}

add()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val += d2.val;
        push(d1);
}

sub()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val -= d2.val;
        push(d1);
}

mul()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val *= d2.val;
        push(d1);
}

div()
{
        Datum d1, d2;
        d2 = pop();
        if (d2.val == 0.0)
                execerror("division by zero", (char *)0);
        d1 = pop();
        d1.val /= d2.val;
        push(d1);
}
```

```
negate()
{
        Datum d;
        d = pop();
        d.val = -d.val;
        push(d);
}
gt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val > d2.val);
        push(d1);
}
lt()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val < d2.val);
        push(d1);
}
ge()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val >= d2.val);
        push(d1);
}
le()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val <= d2.val);
        push(d1);
}
eq()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val == d2.val);
        push(d1);
}
ne()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != d2.val);
        push(d1);
}
and()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
        push(d1);
}
or()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
        push(d1);
}
```

```
not()
{
        Datum d;
        d = pop();
        d.val = (double)(d.val == 0.0);
        push(d);
}

power()
{
        Datum d1, d2;
        extern double Pow();
        d2 = pop();
        d1 = pop();
        d1.val = Pow(d1.val, d2.val);
        push(d1);
}

assign()
{
        Datum d1, d2;
        d1 = pop();
        d2 = pop();
        if (d1.sym->type != VAR && d1.sym->type != UNDEF)
                execerror("assignment to non-variable",
                        d1.sym->name);
        d1.sym->u.val = d2.val;
        d1.sym->type = VAR;
        push(d2);
}

print() /* pop top value from stack, print it */
{
        Datum d;
        d = pop();
        printf("\t%.8g\n", d.val);
}

prexpr()        /* print numeric value */
{
        Datum d;
        d = pop();
        printf("%.8g ", d.val);
}

prstr()         /* print string value */
{
        printf("%s", (char *) *pc++);
}

varread()       /* read into variable */
{
        Datum d;
        extern FILE *fin;
        Symbol *var = (Symbol *) *pc++;
  Again:
        switch (fscanf(fin, "%lf", &var->u.val)) {
        case EOF:
                if (moreinput())
                        goto Again;
                d.val = var->u.val = 0.0;
                break;
        case 0:
                execerror("non-number read into", var->name);
                break;
        default:
                d.val = 1.0;
                break;
        }
        var->type = VAR;
        push(d);
}

Inst *code(f)   /* install one instruction or operand */
        Inst f;
{
        Inst *oprogp = progp;
        if (progp >= &prog[NPROG])
                execerror("program too big", (char *)0);
        *progp++ = f;
        return oprogp;
}
```

```
execute(p)
        Inst *p;
{
        for (pc = p; *pc != STOP && !returning; )
                (*(*pc++))();
}

*****   init.c   ************************************************************

#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double   Log(), Log10(), Sqrt(), Exp(), integer();

static struct {         /* Keywords */
        char    *name;
        int     kval;
} keywords[] = {
        "proc",         PROC,
        "func",         FUNC,
        "return",       RETURN,
        "if",           IF,
        "else",         ELSE,
        "while",        WHILE,
        "print",        PRINT,
        "read",         READ,
        0,              0,
};

static struct {         /* Constants */
        char *name;
        double cval;
} consts[] = {
        "PI",    3.14159265358979323846,
        "E",     2.71828182845904523536,
        "GAMMA", 0.57721566490153286060,  /* Euler */
        "DEG",  57.29577951308232087680,  /* deg/radian */
        "PHI",   1.61803398874989484820,  /* golden ratio */
        0,       0
};

static struct {         /* Built-ins */
        char *name;
        double  (*func)();
} builtins[] = {
        "sin",  sin,
        "cos",  cos,
        "atan", atan,
        "log",  Log,    /* checks range */
        "log10", Log10, /* checks range */
        "exp",  Exp,    /* checks range */
        "sqrt", Sqrt,   /* checks range */
        "int",  integer,
        "abs",  fabs,
        0,      0
};

init()  /* install constants and built-ins in table */
{
        int i;
        Symbol *s;
        for (i = 0; keywords[i].name; i++)
                install(keywords[i].name, keywords[i].kval, 0.0);
        for (i = 0; consts[i].name; i++)
                install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
                s = install(builtins[i].name, BLTIN, 0.0);
                s->u.ptr = builtins[i].func;
        }
}
```

```
*****   math.c   ***************************************************************

#include <math.h>
#include <errno.h>
extern  int     errno;
double  errcheck();

double Log(x)
        double x;
{
        return errcheck(log(x), "log");
}

double Log10(x)
        double x;
{
        return errcheck(log10(x), "log10");
}

double Sqrt(x)
        double x;
{
        return errcheck(sqrt(x), "sqrt");
}

double Exp(x)
        double x;
{
        return errcheck(exp(x), "exp");
}

double Pow(x, y)
        double x, y;
{
        return errcheck(pow(x,y), "exponentiation");
}

double integer(x)
        double x;
{
        return (double)(long)x;
}

double errcheck(d, s)   /* check result of library call */
        double d;
        char *s;
{
        if (errno == EDOM) {
                errno = 0;
                execerror(s, "argument out of domain");
        } else if (errno == ERANGE) {
                errno = 0;
                execerror(s, "result out of range");
        }
        return d;
}

*****   makefile   *************************************************************

YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc6:   $(OBJS)
        cc $(CFLAGS) $(OBJS) -lm -o hoc6

hoc.o code.o init.o symbol.o:   hoc.h

code.o init.o symbol.o: x.tab.h

x.tab.h:        y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr:     hoc.y hoc.h code.c init.c math.c symbol.c
        @pr $?
        @touch pr

clean:
        rm -f $(OBJS) [xy].tab.[ch]
```

Tárgymutató

A

a parancs 24, 331, 334
abd program 203
aktuális jegyzék 38
— sor 333
alburok 94
argumentumok 26, 86
at parancs 48, 145
awk minták 132
— program 115, 131
— változói 135

B

bájt 54
BEGIN minta 133
bejelentkezési jegyzék 34
bemenet átirányítása 43
betűtípusváltó parancs 304
blokkeszközök 79
BREAK billentyű 14
bundle program 111
burok 13, 38, 83, 108
burokciklusok 159
burokfájl 94
— argumentumai 97
burokváltozó 50
burokváltozók 101

C

C-burok 113
c parancs 334
C programozási nyelv 186
cal parancs 148
case utasítás 149
cat parancs 27
cd parancs 37
checkmail program 229
chmod parancs 68, 80
ciklusformák 159
close rendszerhívás 221
cmp parancs 32
comm parancs 121
CONTROL billentyű 14
cp parancs 29, 72
creat rendszerhívás 220
crypt parancs 65
ctl-c billentyű 15
ctl-d billentyű 14, 20
ctl-g billentyű 14
ctl-h billentyű 14
ctl-i billentyű 14
ctl-m billentyű 14
ctl-q karakter 20
ctl-s karakter 20
ctl-u karakter 18

Cs

csoport engedélykészlete 66
csoportazonosító 65
cső (pipe) 44
csőhálózat 45, 84

D

d parancs 334
date parancs 16
dd parancs 121
DELETE billentyű 14
df parancs 80
diction(1) parancs 325
diff parancs 33
doctype program 318
dokumentum 299, 300
du parancs 62

E

e parancs 339
E parancs 339
echo program 40
ed program 24, 330
előregépelés 19
END minta 133
engedélykészlet 64, 66
eqn kulcsszavai 316
— program 300, 313
errno változó 221
error rutin 221

ÉS (AND) feltételes operátor 158
escape karakter 18
eszközfájlok 78
eszközszám 79
execlp rendszerhívás 234
execpv rendszerhívás 234
export parancs 51, 105

F

f parancs 339
fájl megnyitása 217
— vége 57
fájlleíró 105, 217
fájlnév 29
fájlnévrövidítések 38
fájlmutató 192
fájlrendszer 54
— gyökere 64
fájltípus 59
fclose függvény 195
felhasználói azonosító 65
felrakás 79
feltöltés 302
file parancs 58
folyamat 47
folyamatazonosító 47
fopen függvény 192
for utasítás 108
fordított törtvonal 18
fork rendszerhívás 236
formázó programcsomagok 300
fprintf függvény 195
fstat rendszerhívás 229

G

g parancs 335
games parancs 104
get parancs 182
getc függvény 193
getenv rutin 214
global parancsok 336
grep parancs 30

Gy

gyermekfolyamat 61
gyökér 75
gyökérjegyzék 35

H

háttérfolyamat 85
hoc program 246
hoc1 program 246
hoc2 program 254
hoc3 program 258
hoc4 program 269
hoc5 program 277
hoc6 program 284
HOME változó 50

I

i parancs 334
i-bög 70
i-bög-tábla 224
ideal(1) parancs 325
idiff függvény 210
idiff program 208
if utasítás 136
IFS burokváltozó 172
indexbög 69
írás engedély 66
i-számok 70
itt-a-dokumentum 107

J

j parancs 338
jegyzékek 34
jegyzékelem 72
jegyzékengedélyek 68
jelszámok 164
jelszófájl 65
jelzett szabályos kifejezések 338

K

k parancs 338
karaktereszköz 79
karakternevek 308
képernyőszerkesztő 24
keresés 138
keresési út 50
kézikönyvlap 320
kiegyenlítés 302
kijelentkezés 20
kilépési státus 155
kill parancs 47
kimenet átirányítása 43
kivetítendő szövegek 303
különleges karakter 56
külső jelek 239

L

l parancs 332
láncszem 72
learn parancs 23
length függvény 132
levelezés 20

lex program 245, 267
lexikai egység 248
lint program 205
login 16
ls parancs 25
lseek rendszerhívás 222

M

m parancs 337
mail parancs 20
MAIL változó 50
make program 245, 253
man parancs 23
másodlagos promt 89
matematikai képletek 315
megszakításjel 164
mesg parancs 80
metakarakterek 87, 335
mezők 130
mindist függvény 224
mintakarakterek 41
munkajegyzék 34, 61
mv parancs 74

N

naptárprogram 143
newgrp parancs 67
nohup parancs 48
np parancs 332
nroff program 299
null parancssorozat 165
NUMBER 247

Ny

nyomkereső program 203

O

od parancs 55
olvasás engedély 66
opciók 26
opcionális argumentumok 146
open rendszerhívás 220
overwrite 167

Ö

ösvénynév 36

P

p parancs 334
p program 196
parancs 83
parancsargumentumok 99
parancsértelmező 13
parancslezáró 84
parancsösszefűző operátor 158
parse függvény 212
passwd program 67
PATH változó 50
pick parancs 173
pic(1) parancs 325
pozicionális paraméterek 101
pr parancs 28
print utasítás 131
printf függvény 195
profilírozott program 295
programmegállítás 20
promt 16
put parancs 181
putc függvény 193
pwd parancs 35, 61

Q

q parancs 25, 331
Q parancs 331

R

r parancs 339
read rendszerhívás 57
readslow program 218
refer(1) parancs 325
ret függvény 290
RETURN billentyű 14
return utasítás 291
riasztásjel 243
rm parancs 29, 72
roff program 299
root név 65

S

s parancs 334
saját jegyzék 34
sdb program 203
sed program 115
set parancs 102, 122
set-uid bit 67
sh parancs 93
shift parancs 169
SIGKILL jel 206
signal rendszerhívás 239
sizeof függvény 226
sleep függvény 219
— parancs 85
soremelés 57
sorirtó karakter 18
sorok tördelése 141
soron belüli parancsok 310

soronkénti bemenet 57
sort parancs 31, 120
spdist függvény 224
spell(1) parancs 325
spname függvény 223
stat rendszerhívás 229
strlen függvény 199
stty parancs 17, 49
style(1) parancs 325
su parancs 65
substr függvény 132
sv program 230
system rutin 233

SZ

szabályos kifejezések 335
szabványos bemenet 45
— hibakimenet 45
— kimenet 45
számregiszterek 306
szövegformázó program 299
szövegszerkesztő 24
szuperfelhasználó 65
szűrők 115

T

t parancs 337
táblázatok 313
tabs parancs 17
tabulátorok 56
tail parancs 32
társtömbök 139
tbl program 300, 313
tee parancs 84
TERM változó 51
test parancs 155
time parancs 81
titkosított jelszó 66
token 248
tömbök 137
törlőkarakter 18
történetfájl 180
tr parancs 121
trap parancs 164
troff program 299
— tabulátorai 304
tty parancs 80
tulajdonos engedélykészlete 66

U

u parancs 335
újság 22
újságszolgálat 176
újsor-karakter 56
uniq parancs 121
UNIX 9, 13
unlink rendszerhívás 221

V

v parancs 336
VAGY (OR) feltételes operátor 158
varread függvény 292
végrehajtás engedély 66
végrehajtható 67
veremgép 270
veremnyom-kiíratás 204
vis program 187
visszalépés 56
visszhang 14

W

w parancs 25, 331, 339
W parancs 339
wait parancs 47
— rendszerhívás 236
wc parancs 30
who parancs 17, 67
write parancs 21

X

x parancs 339
X parancs 339

Y

yacc program 245, 247

Z

zap program 171

! parancs 21, 332
. (pont) parancs 103
.profile fájl 49
; (pontosvessző) karakter 84
= parancs 338
* (csillag) karakter 86
* (csillag) végjel 117
# metakarakter 90
& jel 47
& (ampersand) jel 85
$ (dollárjel) 16
$* rövidítés 96

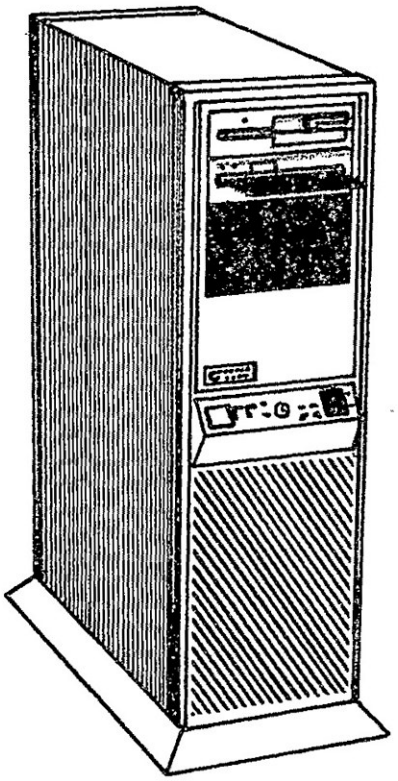
Kiadja a Műszaki Könyvkiadó
Felelős kiadó: Szűcs Péter igazgató
Felelős szerkesztő: Benedikti István
A szedés a Műszaki Könyvkiadóban készült

Készült a VEGYTERV Reprotechnikai üzemében
Felelős vezető: Balogh Mihály
VTV 88.219

Műszaki vezető: Kőrizs Károly
Műszaki szerkesztő: Németh János
A borítót tervezte: Németh János
A könyv formátuma: B5
Ívterjedelme: 32,5 (A5)
Ábrák száma: 5
Betűcsalád és méret: Times 10/11
Azonossági szám: 61 437
MŰ: 4262–i–8891
Készült az MSZ 5601 és 5602 szerint

CONTROLL

A TORONY KIVITELŰ
CONTROLL XT/AT/386-OS SZÁMÍTÓGÉP



ALKALMAS

- HÁLÓZATI SERVERNEK
- GRAFIKUS MUNKAHELYNEK (Workstation)
- IRODAI SZÁMÍTÓGÉPNEK

MERT

- KIS HELYET FOGLAL
- NAGY TELJESÍTMÉNYT NYÚJT
- 6 FÉLMAGAS MEGHAJTÓT FOGAD BE

CONTROLL – EGYETLEN A SOK KÖZÖTT

CONTROLL ELEKTRONIKAI ÉS SZÁMÍTÁSTECHNIKAI KISSZÖVETKEZET
1091 Budapest, Üllői út 101. Tel.: 140-211, 337-392
Bemutatóterem: Budapest IX., Üllői út 101.
Szaküzlet: 1132 Budapest, Visegrádi u. 6. Tel.: 128-064

298 Ft

VIDEOTON

VÁLASZTÉK: OPTIMÁLIS VÁLASZTÁS

VIDEOTON
ELEKTRONIKAI VÁLLALAT

Információ:
SZÁMÍTÁSTECHNIKAI GYÁRA
KERESKEDELMI FŐOSZTÁLY
1062 Budapest, Népköztársaság útja 100.
Telefon: 320-500

A szolgáltatások és korszerű eszközök széles skáláját biztosítják.
16 és 32 bites professzionális személyi számítógépek:
VT 110, VT 160, VT 180.
32 bites szuper mikroszámítógépek:
VT 32, VT 320.